EDB Postgres Distributed (PGD)

Version 5.6

Built at 2024-11-20T17:54:58

1 EDB Postgres Distributed (PGD)

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

Read about why PostgreSQL is better when it's distributed with EDB Postgres Distributed in Distributed PostgreSQL:The Key to Always On Database Availability
PGD Free Trial
Contact Sales

By default, EDB Postgres Distributed uses asynchronous replication, applying changes on the peer nodes only after the local commit. You can configure additional levels of synchronicity between different nodes, groups of nodes, or all nodes by configuring Synchronous Commit, Group Commit (optionally with Eager Conflict Resolution), or CAMO.

Compatibility

EDB Postgres Distributed 5 is compatible with the package versions shown in the table.

Package	Versions
Community PostgreSQL	12-16
EDB Postgres Extended Server	12-16
EDB Postgres Advanced Server	12-16

> **Postgres 16 support**
>
> Postgres 16 support is available only in EDB Postgres Distributed 5.3 and later.

For feature compatibility with compatible servers, see Choosing a Postgres distribution.

2 EDB Postgres Distributed 5.6+ release notes

The EDB Postgres Distributed documentation describes the latest version of EDB Postgres Distributed 5, 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 extension	PGD CLI	PGD Proxy
15 Oct 2024	5.6.0	5.6.0	5.6.0	5.6.0

2.1 EDB Postgres Distributed 5.6.0 release notes

Released: 15 October 2024

EDB Postgres Distributed 5.6.0 includes a number of enhancements and bug fixes.

Highlights

- Improved observability with new monitoring functions and SQL views.
- Improvements to commit scopes including:
  - GROUP COMMIT and SYNCHRONOUS COMMIT support graceful degrading using DEGRADE ON.
  - ORIGIN_GROUP support and commit scope inheritance simplify commit scope creation.
  - Improved synchronous commit behavior around deadlocks.
  - Metrics for commit scope performance and state.

- Optimized Topology support for Subscriber-only groups and nodes. (preview)
- Improved Postgres compliance with support for:
  - Exclusion Constraints
  - REINDEX replications
  - createrole_self_grant
  - column reference in DEFAULT expressions
  - CREATE SCHEMA AUTHORIZATION

- Streaming Transaction support with Decoding Worker.

Enhancements

Component	Version	Description	Addresses
BDR	5.6.0	Decoding Worker supports Streaming Transactions One of the main advantages of streaming is that the WAL sender sends the partial transaction before it commits, which reduces replication lag. Now, with streaming support, the WAL decoder does the same thing, but it streams to the LCRs segments. Eventually, the WAL sender will read the LCRs and mimic the same behavior of streaming large transactions before they commit. This provides the benefits of decoding worker, such as reduced CPU and disk space, as well as the benefits of streaming, such as reduced lag and disk space, since ".spill" files are not generated. The WAL decoder always streams the transaction to LCRs, but based on downstream requests, the WAL sender either streams the transaction or just mimics the normal BEGIN..COMMIT scenario. In addition to the normal LCRs segment files, we create streaming files with the starting names `TR_TXN_<file-name-format>` and `CAS_TXN_<file-name-format>` for each streamed transaction.	

Component	Version	Description	Addresses
BDR	5.6.0	Introduce several new monitoring views There are several view providing new information as well as making some existing information easier to discover: • `bdr.stat_commit_scope` : Cumulative statistics for commit scopes. • `bdr.stat_commit_scope_state` : Information about current use of commit scopes by backends. • `bdr.stat_receiver` : Per subscription receiver statistics. • `bdr.stat_writer` : Per writer statistics. There can be multiple writers for each subscription. This also includes additional information about the currently applied transaction. • `bdr.stat_raft_state` : The state of the Raft consensus on the local node. • `bdr.stat_raft_followers_state` : The state of the followers on the Raft leader node (empty on other nodes), also includes approximate clock drift between nodes. • `bdr.stat_worker` : Detailed information about PGD workers, including what the operation manager worker is currently doing. • `bdr.stat_routing_state` : The state of the connection routing which PGD Proxy uses to route the connections. • `bdr.stat_routing_candidate_state` : Information about routing candidate nodes on the Raft leader node (empty on other nodes).	
BDR	5.6.0	Support conflict detection for exclusion constraints This allows defining `EXCLUDE` constraint on table replicated by PGD either with `CREATE TABLE` or with `ALTER TABLE` and uses similar conflict detection to resolve conflicts as for `UNIQUE` constraints.	
BDR	5.6.0	Detect and resolve deadlocks between synchronous replication wait-for-disconnected sessions and replication writer. This will cancel synchronous replication wait on disconnected sessions if it deadlocks against replication, preventing deadlocks on failovers when using synchronous replication. This only affects commit scopes, not synchronous replication configured via `synchronous_standby_names`.	
BDR	5.6.0	Add bdr.bdr_show_all_file_settings() and bdr.bdr_file_settings view Fix: Correct privileges for bdr_superuser. Creating wrapper SECURITY DEFINER functions in the bdr schema and granting access to bdr_superuser to use those: • bdr.bdr_show_all_file_settings • bdr.bdr_file_settings	
BDR	5.6.0	Add create/drop_commit_scope functions Add functions for creating and dropping commit scopes that will eventually deprecate the non-standard functions for adding and removing commit scopes. Notify the user that these will be deprecated in a future version, suggesting the use of the new versions.	
BDR	5.6.0	Grant additional object permissions to role "bdr_monitor". Permissions for the following objects have been updated to include SELECT permissions for role "bdr_monitor": bdr.node_config	
BDR	5.6.0	Add `bdr.raft_vacuum_interval` and `bdr.raft_vacuum_full_interval` GUCs to control frequency of automatic Raft catalog vacuuming. This update introduces GUCs to regulate the frequency of automatic vacuuming on the specified catalogs. The GUC `bdr.raft_vacuum_interval` determines the frequency at which tables are examined for VACUUM and ANALYZE. Autovacuum GUCs and table reloptions are utilized to ascertain the necessity of VACUUM/ANALYZE. The `bdr.raft_vacuum_full_interval` initiates VACUUM FULL on the tables. Users have the ability to deactivate VACUUM FULL if regular VACUUM suffices to manage bloat.	40412

Component	Version	Description	Addresses
BDR	5.6.0	Add "node_name" to "bdr.node_config_summary" Add "node_name" to the view "bdr.node_config_summary". This makes it consistent with other summary views, which report the name of the object (node, group, etc.) for which the summary is being generated.	
BDR	5.6.0	bdr_init_physical: improve local node connection failure logging Ensure that bdr_init_physical emits details about connection failure if the "--local-dsn" parameter is syntactically correct but invalid, e.g., due to an incorrect host or port setting.	
BDR	5.6.0	`bdr_config` : add PG_FLAVOR output `bdr_config` now shows the PostgreSQL "flavor" which BDR was built against, one of: • COMMUNITY • EPAS • EXTENDED • BDRPG	
BDR	5.6.0	Enhance warning messages Enhance messages issued during DML and DDL lock acquisition.	
BDR	5.6.0	Do not send Raft snapshot very aggressively Avoid sending Raft snapshots too frequently as it can slow down follower nodes. Limit the snapshot rate to once in every election timeout, unless there is no other communication between the nodes, in which case send a snapshot every 1/3rd of the election timeout. This will help all nodes keep pace with the leader and improve CPU utilization.	37725
BDR	5.6.0	Group-Specific Configuration Options It is now possible to set akk top-level and subgroup level options. The following options are available for top-both groups: • check_constraints • enable_wal_decoder • num_writers • streaming_mode • enable_raft Subgroups inherit settings from their parent group, but can override them if set in the subgroup.	37725
BDR	5.6.0	Subscriber-only node groups have a leader Subscriber-only node groups have a leader elected by top-level Raft. There is now a bdr.leader catalog that tracks leadership of subgroups and subscriber-only nodes. If the node that is the leader of a subscriber-only node group goes down or becomes unreachable, a new leader is elected from that group.	
BDR	5.6.0	Optimized topology for subscriber-only nodes via the leader of the subscriber-only node group Subscriber-only nodes earlier used to have subscriptions to each data node. Now if optimized topology is enabled, only the leaders of subscriber-only node groups have subscriptions to routing leaders of data node subsgroups. The subscriber only nodegroup leaders route data to other nodes of that subscriber-only nodegroup. This reduces the load on all data nodes so they do not have to send data to all subscriber-only nodes. The GUC `bdr.force_full_mesh=off` enables this optimized topology. This GUC variable is on by default, retaining pre-5.6.0 behavior.	
BDR	5.6.0	Introduce new subscription types to support optimized topology New subscription types that forward data from all nodes of the subgroup via a routing leader (mode: l), and those that forward data from the entire cluster via a subscriber-only group leader (mode: w) are introduced.	

Component	Version	Description	Addresses
BDR	5.6.0	Introduce version number and timestamp for write leader A write leader has a version. Every time a new leader is elected, the version is incremented and timestamp noted via Raft. This is to build a foundation for better conflict resolution.	
BDR	5.6.0	Allow use of column reference in DEFAULT expressions Using column references in default expressions is now supported, this is particularly useful with generated columns, for example: `ALTER TABLE gtest_tableoid ADD COLUMN c regclass GENERATED ALWAYS AS (tableoid) STORED;`	
BDR	5.6.0	Support replication of REINDEX Both REINDEX and REINDEX CONCURRENTLY are now replicated commands.	
BDR	5.6.0	Fix receiver worker being stuck when exiting Receiver worker could get stuck when exiting, waiting for a writer that never actually started. This could on rare occasions break replication after configuration changes until Postgres was restarted.	
BDR	5.6.0	Reduce performance impact of PGD specific configuration parameters that are sent to client Changes to values of variables `bdr.last_committed_lsn` , `transaction_id` and `bdr.local_node_id` are automatically reported to clients when using CAMO or GROUP COMMIT. This has now been optimized to use less resources.	
BDR	5.6.0	Allow use of commit scopes defined in parent groups When there is a commit scope defined for top-level group, it can be used by any node in a subgroup and does not need to be redefined for every subgroup anymore. This is particularly useful when combined with `ORIGIN\_GROUP` keyword to reduce the complexity of commit scope setup.	
PGD CLI	5.6.0	Use bdr.bdr_file_settings view in verify-settings Use bdr.bdr_file_settings view to get the current settings for the proxy.	

Bug Fixes

Component	Version	Description	Addresses
BDR	5.6.0	Fixed buffer overrun in the writer Include an extra zero byte at the end of a column value allocation in shared memory queue insert/update/delete messages.	98966
BDR	5.6.0	Fixes for some race conditions to prevent node sync from entering a hung state with the main subscription disabled.	
BDR	5.6.0	Do not accidentally drop the autopartition rule when a column of the autopartitioned table is dropped. When ALTER TABLE .. DROP COLUMN is used, the object_access_hook is fired with `classId` set to RelationRelationId, but the `subId` is set to the attribute number to differentiate it from the DROP TABLE command. Therefore, we need to check the subId field to make sure that we are not performing actions that should only be triggered when a table is dropped.	40258

Component	Version	Description	Addresses
BDR	5.6.0	Adjust `bdr.alter_table_conflict_detection()` to propagate correctly to all nodes Ensure that the propagation of `bdr.alter_table_conflict_detection()` (as well as the related, deprecated `bdr.column_timestamps_(en|dis)able()` functions) is carried out correctly to all logical standbys. Previously, this propagation did not occur if the logical standby was not directly attached to the node on which the functions were executed.	40258
BDR	5.6.0	Prevent a node group from being created with a duplicate name Ensure that a nodegroup is not inadvertently created with the same name as an existing nodegroup. Failure to do so may result in a complete shutdown of the top-level Raft on all nodes, with no possibility of recovery.	
BDR	5.6.0	Prevent spurious "local info ... not found" errors when parting nodes Handle the absence of the expected node record gracefully when a node is being removed, the local node record might have already been deleted, but an attempt could be made to update it anyway. This resulted in harmless "BDR node local info for node ... not found" errors.	
BDR	5.6.0	Prevent a corner-case situation from being misdiagnosed as a PGD version problem Improve Raft error messages to handle cases where nodes may not be correctly participating in Raft.	
BDR	5.6.0	Handling duplicate requests in RAFT preventing protocol breakage When processing RAFT entries, it's crucial to handle duplicate requests properly to prevent Raft protocol issues. Duplicate requests can occur when a client retries a request that has already been accepted and applied by the Raft leader. The problem arose when the leader failed to detect the duplicate request due to historical evidence being pruned.	37725
BDR	5.6.0	Handling Raft Snapshots: Consensus Log When installing or importing a Raft snapshot, discard the consensus log unless it contains an entry matching the snapshot's last included entry and term.	37725
BDR	5.6.0	Be more restrictive about which index to use during replication for REPLICA IDENTITY FULL tables This fixes various index related errors during replication like: 'could not lookup equality operator for type, optype in opfamily' or 'function "amgettuple" is not defined for index "brinidx"'	
BDR	5.6.0	Support `createrole_self_grant` The `createrole_self_grant` configuration option affects inherited grants by newly created roles. In previous versions `CREATE ROLE` / `CREATE USER` replication would not take this into consideration, resulting in different role privileges on different nodes.	
BDR	5.6.0	Allow `CREATE SCHEMA AUTHORIZATION ...` combined with other create operations Previously, this would throw "cannot change current role within security-restricted operation" error	
BDR	5.6.0	Use base type instead of domain type while casting values This prevents errors when replicating UPDATEs for domains defined as NOT VALID where tables contain data which would not be allowed by current definition of such domain.	
Utilities	5.6.0	bdr_pg_upgrade - Create logical slot with twophase set to true for PG 14+	

3 Known issues

These are currently known issues in EDB Postgres Distributed 5. These known issues are tracked in PGD's ticketing system and are expected to be resolved in a future release.

- 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 PGD writer. As mentioned in Conflicts, `skip` isn't 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.

- Lag Control doesn't adjust commit delay in any way on a fully isolated node, that's in case all other nodes are unreachable or not operational. As soon as at least one node connects, replication Lag Control picks up its work and adjusts the PGD commit delay again.

- For time-based Lag Control, PGD currently uses the lag time, measured by commit timestamps, rather than the estimated catch up time that's based on historic apply rates.

- 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 require a restart of Postgres or even a reload of the configuration.

- Group Commit can't be combined with CAMO.

- Transactions using Eager Replication can't yet execute DDL. The TRUNCATE command is allowed.

- Parallel Apply isn't currently supported in combination with Group Commit. Make sure to disable it when using Group Commit by either (a) Setting `num_writers` to 1 for the node group using `bdr.alter_node_group_config` or (b) using 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's 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. Make sure that any transactions using a specific commit scope have finished before altering or removing it.

- The PGD CLI can return stale data on the state of the cluster if it's still connecting to nodes that were previously parted from the cluster. Edit the `pgd-cli-config.yml` file, or change your `--dsn` settings to ensure only active nodes in the cluster are listed for connection.

To modify a commit scope safely, use `bdr.alter_commit_scope`.

- Tables configured with `REPLICA IDENTITY FULL` and include `box`, `polygon` or `json` types in their columns are using PGD aren't able to be replicated. You can mitigate this issue by configuring a primary key for `REPLICA IDENTITY` to use or, for `json` columns only, using the `jsonb` type instead.

Details of other design or implementation limitations are also available.

4 Terminology

This terminology list includes terms associated with EDB Postgres Distributed that you might be unfamiliar with.

Asynchronous replication

A type of replication that copies data to other PGD cluster members after the transaction completes on the origin node. Asynchronous replication can provide higher performance and lower latency than synchronous replication. However, asynchronous replication can see a lag in how long changes take to appear in the various cluster members. While the cluster will be eventually consistent, there's potential for nodes to be apparently out of sync with each other.

Commit scopes

Rules for managing how transactions are committed between the nodes and groups of a PGD cluster. Used to configure synchronous replication, Group Commit, CAMO, Eager, Lag Control, and other PGD features.

CAMO or commit-at-most-once

High-value transactions in some applications require that the application successfully commits exactly once, and in the event of failover and retrying, only once. To ensure this happens in PGD, CAMO can be enabled, allowing the application to actively participate in the transaction.

Conflicts

As data is replicated across the nodes of a PGD cluster, there might be occasions when changes from one source clash with changes from another source. This is a conflict and can be handled with conflict resolution. (Conflict resolution is a set of rules that decide which source is correct or preferred.) Conflicts can also be avoided with conflict-free data types.

Consensus

How Raft makes group-wide decisions. Given a number of nodes in a group, Raft looks for a consensus of the majority (number of nodes divided by 2 plus 1) voting for a decision. For example, when a write leader is being selected, a Raft consensus is sought over which node in the group will be the write leader. Consensus can be reached only if there's a quorum of voting members.

Cluster

Generically, a cluster is a group of multiple redundant systems arranged to appear to end users as one system. See also PGD cluster and Postgres cluster.

DDL (data definition language)

The subset of SQL commands that deal with defining and managing the structure of a database. DDL statements can create, modify, and delete objects (that is, schemas, tables, and indexes) in the database. Common DDL commands are CREATE, ALTER, and DROP.

DML (data manipulation language)

The subset of SQL commands that deal with manipulating the data held in a database. DML statements can create, modify, and delete rows in tables in the database. Common DML commands are INSERT, UPDATE, and DELETE.

Eager

A synchronous commit mode that avoids conflicts by detecting incoming potentially conflicting transactions and “eagerly” aborts one of them to maintain consistency.

Eventual consistency

A distributed computing consistency model stating changes to the same item in different cluster members will eventually converge to the same value. Asynchronous logical replication with conflict resolution and conflict-free replicated data types exhibit eventual consistency in PGD.

Failover

The automated process that recognizes a failure in a highly available database cluster and takes action to maintain consistency and availability. The goal is to minimize downtime and data loss.

Group commit

A synchronous commit mode that requires more than one PGD node to successfully receive and confirm a transaction at commit time.

Immediate consistency

A distributed computing model where all replicas are updated synchronously and simultaneously. This model ensures that all reads after a write completes will see the same value on all nodes. The downside of this approach is its negative impact on performance.

Logical replication

A more efficient method of replicating changes in the database. While physical streaming replication duplicates the originating database’s disk blocks, logical replication instead takes the changes made, independent of the underlying physical storage format, and publishes them to all systems that subscribed to see the changes. Each subscriber then applies the changes locally. Logical replication can't support most DDL commands.

Node

A general term for an element of a distributed system. A node can play host to any service. In PGD, PGD nodes run a Postgres database, the BDR extension, and optionally a PGD Proxy service.

Typically, for high availability, each node runs on separate physical hardware, but that's not always the case. For example, a proxy might share a hardware node with a database.

Node groups

PGD nodes in PGD clusters can be organized into groups to reflect the logical operation of the cluster. For example, the data nodes in a particular physical location can be part of a dedicated node group for the location.

PGD cluster

A group of multiple redundant database systems and proxies arranged to avoid single points of failure while appearing to end users as one system. PGD clusters can be run on Docker instances, cloud instances or “bare” Linux hosts, or a combination of those platforms. A PGD cluster can also include backup and proxy nodes. The data nodes in a cluster are grouped together in a top-level group and into various local node groups.

PGD node

In a PGD cluster are nodes that run databases and participate in the PGD cluster. A typical PGD node runs a Postgres database, the BDR extension, and optionally a PGD Proxy service. PGD modes are also referred to as *data nodes*, which suggests they store data. However, some PGD nodes, specifically witness nodes, don't do that.

Physical replication

By making an exact copy of database disk blocks as they're modified to one or more standby cluster members, physical replication provides an easily implemented method to replicate servers. But there are restrictions on how it can be used. For example, only one master node can run write transactions. Also, the method requires that all cluster members are on the same major version of the database software with the same operating system and CPU architecture.

Postgres cluster

Traditionally, in PostgreSQL, a number of databases running on a single server is referred to as a cluster (of databases). This kind of Postgres cluster isn't highly available. To get high availability and redundancy, you need a PGD cluster.

Quorum

A quorum is the minimum number of voting nodes needed to participate in a distributed vote. It ensures that the decision made has validity. For example, when a Raft consensus is needed by a PGD cluster, a minimum number of voting nodes participating in the vote are needed. With a 5-node cluster, the quorum is 3 nodes in the cluster voting. A consensus is 5/2+1 nodes, 3 nodes voting the same way. If there are only 2 voting nodes, then a consensus is never established. Quorums are required in PGD for global locks and Raft decisions.

Replicated available fault tolerance (Raft)

A consensus algorithm that uses votes from a quorum of machines in a distributed cluster to establish a consensus. PGD uses Raft within groups (top-level or local) to establish the node that's the write leader.

Read scalability

The ability of a system to handle increasing read workloads. For example, PGD can introduce 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.

Subscription

PGD nodes will publish changes being made to data to nodes that are interested. Other PGD nodes will ask to subscribe to those changes. This behavior creates a subscription and is the mechanism by which each node is updated. PGD nodes bidirectionally subscribe to other PGD nodes' changes.

Switchover

A planned change in connection between the application or proxies 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 a two-phase commit. While this approach replicates changes and resolves conflicts before committing, a performance cost in latency occurs due to the coordination required across nodes.

Subscriber-only nodes

A PGD cluster is based around bidirectional replication. But in some use cases, such as needing a read-only server, bidirectional replication isn't needed. A subscriber-only node is used in this case. It subscribes only to changes in the database to keep itself up to date and provide correct results to any run directly on the node. This feature can be used to enable horizontal read scalability in a PGD cluster.

Two-phase commit (2PC)

A multi-step process for achieving consistency across multiple database nodes. The first phase sees a transaction prepared on an originating node and sent to all participating nodes. Each participating node validates that it can apply the transaction and signals its readiness to the originating node. This is the prepare phase. In the second phase, if all the participating nodes signal they're ready, the originating node proceeds to commit the transaction and signals the participating nodes to commit, too. This is the commit phase. If, in the prepare phase, any node signals it isn't ready, the entire transaction is aborted. This process ensures all nodes get the same changes.

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, for example, Oracle Exadata.

Witness nodes

Witness nodes primarily serve to help the cluster establish a consensus. An odd number of data nodes is needed to establish a consensus. Where resources are limited, a witness node can be used to participate in cluster decisions but not replicate the data. Not holding the data means it can't operate as a standby server or provide majorities in synchronous commits.

Write leader

In an Always-on architecture, a node is selected as the correct connection endpoint for applications. This node is called the write leader. Once selected, proxy nodes route queries and updates to it. With only one node receiving writes, unintended multi-node writes can be avoided. The write leader is selected by consensus of a quorum of data nodes. If the write leader becomes unavailable, the data nodes select another node to become write leader. Nodes that aren't the write leader are referred to as *shadow nodes*.

Writer

When a subscription delivers data changes to a PGD node, the database server tasks a worker process, called a writer, with getting those changes applied.

5 PGD overview

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

Read about why PostgreSQL is better when it's distributed with EDB Postgres Distributed in Distributed PostgreSQL:The Key to Always On Database Availability
PGD Free Trial
Contact Sales

PGD provides loosely coupled, multimaster 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 PGD group.

By default, PGD 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 PGD node is a member of at least one *node group*. In the most basic architecture, there's a single node group for the whole PGD cluster.

Multiple masters

Each node (database) participating in a PGD 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 called the write leader.

Asynchronous, by default

Changes made on one PGD 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 PGD, there are multiple masters and, as a result, multiple data streams. So data on different nodes might differ even when `synchronous_commit` and `synchronous_standby_names` are used.

Mesh topology

PGD 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 PGD except in special circumstances, such as adding and removing nodes. Data can arrive from outside the EDB Postgres Distributed cluster or be sent onward 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's done, later commits are sent to other nodes as they occur in real time. Changes are replicated without executing SQL again, 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.

Connection management

Connection management leverages consensus-driven quorum to determine the correct connection endpoint in a semi-exclusive manner to prevent unintended multi-node writes from an application. This approach reduces the potential for data conflicts. The node selected as the correct connection endpoint at any point in time is referred to as the write leader.

PGD Proxy is the tool for application connection management provided as part of EDB Postgres Distributed.

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 is a logical standby node. (Postgres physical standbys aren't supported by PGD.)

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 a 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 locations. Each group provides high availability. 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 selected as the group's write leader. Proxies then direct application writes and queries to the write leader. The other nodes are replicas of the write leader. If, at any point, the write leader is seen to be unavailable, the remaining nodes in the group select a new write leader from the group the proxies direct traffic to that node. Scalability isn't the goal of this architecture.

Since writes are mainly to only one node, the possibility of contention between nodes 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 Postgres database servers

PGD is compatible with PostgreSQL, EDB Postgres Extended Server, and EDB Postgres Advanced Server and is deployed as a standard Postgres extension named BDR. See Compatibility for details about supported version combinations.

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

PGD 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 you can safely replicate in a multi-node setting. Application usage goes into detail about how PGD behaves from an application development perspective.

Characteristics affecting performance

By default, PGD 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 and is immune to replication conflicts on other nodes caused by long-running SELECT queries. 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 multimaster technology and aren't particular to PGD.

Synchronous replication 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 PGD group doesn't result in significant write throughput increase when most tables are replicated because all the writes are replayed on all nodes. Because PGD writes are in general more effective than writes coming from Postgres clients by way of SQL, you can increase performance. Read throughput generally scales linearly with the number of nodes.

6 Introducing PGD quick starts

Quick start

EDB Postgres Distributed (PGD) is a multi-master replicating implementation of Postgres designed for high performance and availability. You can create database clusters made up of many bidirectionally synchronizing database nodes. The clusters can have a number of proxy servers that direct your query traffic to the most available nodes, adding further resilience to your cluster configuration.

Other deployment options

- If you prefer to have a fully managed EDB Postgres Distributed experience, PGD is now available as an option on BigAnimal, EDB's cloud platform for Postgres. See BigAnimal distributed high-availability clusters.
- If you prefer to deploy PGD on Kubernetes, you can use the EDB PGD Operator for Kubernetes. See EDB PGD Operator for Kubernetes.

What's in this quick start

PGD is very configurable. To quickly evaluate and deploy PGD, use this quick start. It'll get you up and running with a fully configured PGD cluster using the same tools that you'll use to deploy to production. This quick start includes:

- A short introduction to Trusted Postgres Architect (TPA) and how it helps you configure, deploy, and manage EDB Postgres Distributed
- A guide to selecting Docker, Linux hosts, or AWS quick starts
  - The Docker quick start
  - The Linux host quick start
  - The AWS quick start
- Connecting applications to your cluster
- Further explorations with your cluster including
  - Replication
  - Conflicts
  - Failover

Introducing PGD and TPA

PGD is a multi-master replicating implementation of Postgres designed for high performance and availability. The installation of PGD is orchestrated by TPA.

We created TPA to make installing and managing various Postgres configurations easily repeatable. TPA orchestrates creating and deploying Postgres.

These quick starts are designed to let you quickly get a single region cluster.

In these quick starts, you install TPA first. If you already have TPA installed, you can skip those steps. TPA is more of a tool than a simple installer. You can use the same installation of TPA to deploy many different configurations of Postgres clusters.

You'll use TPA to generate a configuration file for a PGD demonstration cluster. This cluster will have three replicating database nodes, cohosting three high-availability proxies and one backup node.

You will then use TPA to provision and deploy the required configuration and software to each node.

Selecting Docker, Linux hosts, or AWS quick starts

Three quick starts are currently available:

- Docker — Provisions, deploys, and hosts the cluster on Docker containers on a single machine.
- Linux hosts — Deploys and hosts the cluster on Linux servers that you already provisioned with an operating system and SSH connectivity. These can be actual physical servers or virtual machines, deployed on-premises or in the cloud.
- AWS — Provisions, deploys, and hosts the cluster on AWS.

Docker quick start

The Docker quick start is ideal for those looking to initially explore PGD and its capabilities. This configuration of PGD isn't suitable for production use but can be valuable for testing the functionality and behavior of PGD clusters. You might also find it useful when familiarizing yourself with PGD commands and APIs to prepare for deploying on cloud, VM, or Linux hosts.

- Begin the Docker quick start.

Linux host quick start

The Linux hosts quick start is suited if you intend to install PGD on your own hosts, where you have complete control of the hardware and software, or in a private cloud. The overall configuration is similar to the Docker configuration but is more persistent over system restarts and closer to a single-region production deployment of PGD.

- Begin the Linux host quick start.

AWS quick start

The AWS quick start is more extensive and deploys the PGD cluster onto EC2 nodes on Amazon's cloud. The cluster's overall configuration is similar to the Docker quick start. However, instead of using Docker containers, it uses t3.micro instances of Amazon EC2 to provide the compute power. The AWS deployment is more persistent and not subject to the limitations of the Docker quick start deployment. However, it requires more initial setup to configure the AWS CLI.

- Begin the AWS quick start.

Further explorations with your cluster

- Connect applications to your PGD cluster.
- Find out how a PGD cluster stands up to downtime of data nodes or proxies.
- Learn about how EDB Postgres Distributed manages conflicting updates.
- Move beyond the quick starts.

6.1 Deploying an EDB Postgres Distributed example cluster on Docker

This quick start uses TPA to set up PGD with an Always-on Single Location architecture using local Docker containers.

Introducing TPA and PGD

We created TPA to make installing and managing various Postgres configurations easily repeatable. TPA orchestrates creating and deploying Postgres. In this quick start, you install TPA first. If you already have TPA installed, you can skip those steps. You can use TPA to deploy various configurations of Postgres clusters.

PGD is a multi-master replicating implementation of Postgres designed for high performance and availability. The installation of PGD is orchestrated by TPA. You will use TPA to generate a configuration file for a PGD demonstration cluster.

This cluster uses local Docker containers to host the cluster's nodes: three replicating database nodes, three cohosted connection proxies, and one backup node. You can then use TPA to provision and deploy the required configuration and software to each node.

This configuration of PGD isn't suitable for production use but can be valuable for testing the functionality and behavior of PGD clusters. You might also find it useful when familiarizing yourself with PGD commands and APIs to prepare for deployment on cloud, VM, or Linux hosts.

> **Note**
>
> This set of steps is specifically for Ubuntu 22.04 LTS on Intel/AMD processors.

Prerequisites

To complete this example, you need a system with enough RAM and free storage. You also need curl and Docker installed.

RAM requirements

You need a minimum of 4GB of RAM on the system. You need this much RAM because you will be running four containers, three of which will be hosting Postgres databases.

Free disk space

You need at least 5GB of free storage, accessible by Docker, to deploy the cluster described by this example. We recommend that you have a bit more than that.

The curl utility

You will download and run scripts during this quick start using the curl utility, which might not be installed by default. To ensure that curl is installed, run:

```
sudo apt update
sudo apt install curl
```

Docker Engine

You will use Docker containers as the target platform for this PGD deployment. Install Docker Engine:

```
sudo apt update
sudo apt install docker.io
```

> **Running as a non-root user**
>
> Once Docker Engine is installed, be sure to add your user to the Docker group:
>
> ```
> sudo usermod -aG docker <username>
> newgrp docker
> ```

Preparation

EDB account

To install both TPA and PGD, you need an EDB account.

Sign up for a free EDB account if you don't already have one. Signing up gives you a trial subscription to EDB's software repositories.

After you're registered, go to the EDB Repos 2.0 page, where you can obtain your repo token.

On your first visit to this page, select **Request Access** to generate your repo token. Copy the token using the **Copy Token** icon, and store it safely.

Setting environment variables

First, set the `EDB_SUBSCRIPTION_TOKEN` environment variable to the value of your EDB repo token, obtained in the EDB account step.

```
export EDB_SUBSCRIPTION_TOKEN=<your-repo-token>
```

You can add this to your `.bashrc` script or similar shell profile to ensure it's always set.

Configure the repository

All the software needed for this example is available from the EDB Postgres Distributed package repository. The following command downloads and runs a script to configure the EDB Postgres Distributed repository. This repository also contains the TPA packages.

```
curl -1sLf "https://downloads.enterprisedb.com/$EDB_SUBSCRIPTION_TOKEN/postgres_distributed/setup.deb.sh" | sudo
-E bash
```

> **Troubleshooting repo access**
>
> The script should produce output starting with:
>
> ```
> Executing the  setup script for the 'enterprisedb/postgres_distributed' repository ...
> ```
>
> If it produces no output or an error, double-check that you entered your token correctly. It the problem persists, contact Support for assistance.

Installing Trusted Postgres Architect (TPA)

You'll use TPA to provision and deploy PGD. If you previously installed TPA, you can move on to thenext step. You'll find full instructions for installing TPA in the Trusted Postgres Architect documentation, which we've also included here.

Linux environment

TPA supports several distributions of Linux as a host platform. These examples are written for Ubuntu 22.04, but steps are similar for other supported platforms.

Install the TPA package

```
sudo apt install tpaexec
```

Configuring TPA

You now need to configure TPA, which configures TPA's Python environment. Call `tpaexec` with the command `setup`:

```
sudo /opt/EDB/TPA/bin/tpaexec setup
export PATH=$PATH:/opt/EDB/TPA/bin
```

You can add the `export` command to your shell's profile.

Testing the TPA installation

You can verify TPA is correctly installed by running `selftest`:

```
tpaexec selftest
```

TPA is now installed.

Installing PGD using TPA

Generating a configuration file

Run the `tpaexec configure` command to generate a configuration folder:

```
tpaexec configure democluster \
  --architecture PGD-Always-ON \
  --platform docker \
  --edb-postgres-advanced 16 \
  --redwood \
  --location-names dc1 \
  --pgd-proxy-routing local \
  --no-git \
  --hostnames-unsorted \
  --keyring-backend legacy
```

You specify the PGD-Always-ON architecture (`--architecture PGD-Always-ON`), which sets up the configuration for PGD's Always-on architectures. As part of the default architecture, it configures your cluster with three data nodes, cohosting three PGD Proxy servers, along with a Barman node for backup.

Specify that you're using Docker (`--platform docker`). By default, TPA configures Rocky Linux as the default image for all nodes.

> **Deployment platforms**
>
> Other Linux platforms are supported as deployment targets for PGD. See the EDB Postgres Distributed compatibility table for details.
>
> Observe that you don't have to deploy PGD to the same platform you're using to run TPA!

Specify that the data nodes will be running EDB Postgres Advanced Server v16 (`--edb-postgres-advanced 16`) with Oracle compatibility (`--redwood`).

You set the notional location of the nodes to `dc1` using `--location-names`. You then set `--pgd-proxy-routing` to `local` so that proxy routing can route traffic to all nodes in each location.

By default, TPA commits configuration changes to a Git repository. For this example, you don't need to do that, so pass the `--no-git` flag.

You also ask TPA to generate repeatable hostnames for the nodes by passing `--hostnames-unsorted`. Otherwise, it selects hostnames at random from a predefined list of suitable words.

Finally, `--keyring-backend legacy` tells TPA to select the legacy version of the keyring backend. Secrets are stored with an older keyring backend, as the version of Ubuntu this example is based on doesn't support the newer keyring backend.

This command creates a subdirectory called `democluster` in the current working directory. It contains the `config.yml` configuration file TPA uses to create the cluster. You can view it using:

```
less democluster/config.yml
```

> **Further reading**
>
> - View the full set of available options by running:
>
>   ```
>   tpaexec configure --architecture PGD-Always-ON --help
>   ```
>
> - More details on PGD-Always-ON configuration options in Deploying with TPA
> - PGD-Always-ON in the TPA documentation
> - `tpaexec configure` in the TPA documentation
> - Docker platform in the TPA documentation

Deploying the cluster

You can now deploy the distributed cluster. For Docker deployments, deploying both provisions the required Docker containers and deploys the software to those containers:

```
tpaexec deploy democluster
```

TPA applies the configuration, installing the needed packages and setting up the actual EDB Postgres Distributed cluster.

> **Further reading**
>
> - `tpaexec deploy` in the Trusted Postgres Architect documentation

Connecting to the cluster

You're now ready to log in to one of the nodes of the cluster with SSH and then connect to the database. Part of the configuration process is to set up SSH logins for all the nodes, complete with keys. To use the SSH configuration, you need to be in the `democluster` directory created by the `tpaexec configure` command earlier:

```
cd democluster
```

From there, you can run `ssh -F ssh_config <hostname>` to establish an SSH connection. You will connect to kaboom, the first database node in the cluster:

```
ssh -F ssh_config kaboom
```

output

```
[root@kaboom ~]#
```

Notice that you're logged in as `root` on `kaboom`.

You now need to adopt the identity of the enterprisedb user. This user is preconfigured and authorized to connect to the cluster's nodes.

```
sudo -iu enterprisedb
```

output

```
enterprisedb@kaboom:~ $
```

You can now run the `psql` command to access the `bdrdb` database:

```
psql bdrdb
```

output

```
psql (16.2.0, server 16.2.0)
Type "help" for help.

bdrdb=#
```

You're directly connected to the Postgres database running on the kaboom node and can start issuing SQL commands.

To leave the SQL client, enter `exit`.

Using PGD CLI

The pgd utility, also known as the PGD CLI, lets you control and manage your EDB Postgres Distributed cluster. It's already installed on the node.

You can use it to check the cluster's health by running `pgd check-health`:

```
pgd check-health
```

output

```
 Check      Status Message
 -----      ------ -------
 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
enterprisedb@kaboom:~ $
```

Or, you can use `pgd show-nodes` to ask PGD to show you the data-bearing nodes in the cluster:

```
pgd show-nodes
```

output

```
 Node    Node ID    Group        Type Current State Target State Status Seq ID
 ----    -------    -----        ---- ------------- ------------ ------ ------
kaboom 2710197610 dc1_subgroup data ACTIVE        ACTIVE       Up     1
kaftan 3490219809 dc1_subgroup data ACTIVE        ACTIVE       Up     3
kaolin 2111777360 dc1_subgroup data ACTIVE        ACTIVE       Up     2
enterprisedb@kaboom:~ $
```

Similarly, use `pgd show-proxies` to display the proxy connection nodes:

```
pgd show-proxies
```

output

```
 Proxy  Group        Listen Addresses Listen Port
 -----  -----        ---------------- -----------
kaboom dc1_subgroup [0.0.0.0]        6432
kaftan dc1_subgroup [0.0.0.0]        6432
kaolin dc1_subgroup [0.0.0.0]        6432
```

The proxies provide high-availability connections to the cluster of data nodes for applications. You can connect to the proxies and, in turn, to the database with the command `psql -h kaboom,kaftan,kaolin -p 6432 bdrdb`:

```
psql -h kaboom,kaftan,kaolin -p 6432 bdrdb
```

output

```
 psql (16.2.0, server 16.2.0)
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, compression: off)
Type "help" for help.

bdrdb=#
```

Explore your cluster

- Connect to your database to applications.
- Explore replication with hands-on exercises.
- Explore failover with hands-on exercises.
- Understand conflicts by creating and monitoring them.
- Take the next steps for working with your cluster.

6.2 Deploying an EDB Postgres Distributed example cluster on Linux hosts

Introducing TPA and PGD

We created TPA to make installing and managing various Postgres configurations easily repeatable. TPA orchestrates creating and deploying Postgres. In this quick start, you install TPA first. If you already have TPA installed, you can skip those steps. You can use TPA to deploy various configurations of Postgres clusters.

PGD is a multi-master replicating implementation of Postgres designed for high performance and availability. The installation of PGD is orchestrated by TPA. You will use TPA to generate a configuration file for a PGD demonstration cluster.

The TPA Linux host option allows users of any cloud or VM platform to use TPA to configure EDB Postgres Distributed. All you need from TPA is for the target system to be configured with a Linux operating system and accessible using SSH. Unlike the other TPA platforms (Docker and AWS), the Linux host configuration doesn't provision the target machines. You need to provision them wherever you decide to deploy.

This cluster uses Linux server instances to host the cluster's nodes. The nodes include three replicating database nodes, three cohosted connection proxies, and one backup node. TPA can then provision, prepare, and deploy the required EDB Postgres Distributed software and configuration to each node.

> **On host compatibility**
>
> This set of steps is specifically for users running Ubuntu 22.04 LTS on Intel/AMD processors.

Prerequisites

Configure your Linux hosts

You need to provision four hosts for this quick start. Each host must have a supported Linux operating system installed. To eliminate prompts for password, each host also needs to be SSH accessible using certificate key pairs.

> **On machine provisioning**
>
> Azure users can follow a Microsoft guide on how to provision Azure VMs loaded with Linux. Google Cloud Platform users can follow a Google guide on how to provision GCP VMs with Linux loaded. You can use any virtual machine technology to host a Linux instance, too. Refer to your virtualization platform's documentation for instructions on how to create instances with Linux loaded on them.
>
> Whichever cloud or VM platform you use, you need to make sure that each instance is accessible by SSH and that each instance can connect to the other instances. They can connect through either the public network or over a VPC for the cloud platforms. You can connect through your local network for on-premises VMs.
>
> If you can't do this, you might want to consider the Docker or AWS quick start. These configurations are easier to set up and quicker to tear down. The AWS quick start, for example, automatically provisions compute instances and creates a VPC for those instances.

In this quick start, you will install PGD nodes onto four hosts configured in the cloud. Each of these hosts in this example is installed with Rocky Linux. Each has a public IP address to go with its private IP address.

Host name	Public IP	Private IP
linuxhost-1	172.19.16.27	192.168.2.247
linuxhost-2	172.19.16.26	192.168.2.41
linuxhost-3	172.19.16.25	192.168.2.254
linuxhost-4	172.19.16.15	192.168.2.30

These are example IP addresses. Substitute them with your own public and private IP addresses as you progress through the quick start.

Set up a host admin user

Each machine requires a user account to use for installation. For simplicity, use a user with the same name on all the hosts. On each host, also configure the user so that you can SSH into the host without being prompted for a password. Be sure to give that user sudo privileges on the host. On the four hosts, the user rocky is already configured with sudo privileges.

Preparation

EDB account

You need an EDB account to install both TPA and PGD.

Sign up for a free EDB account if you don't already have one. Signing up gives you a trial subscription to EDB's software repositories.

After you're registered, go to the EDB Repos 2.0 page, where you can obtain your repo token.

On your first visit to this page, select **Request Access** to generate your repo token. Copy the token using the **Copy Token** icon, and store it safely.

Setting environment variables

First, set the `EDB_SUBSCRIPTION_TOKEN` environment variable to the value of your EDB repo token, obtained in the EDB account step.

```
export EDB_SUBSCRIPTION_TOKEN=<your-repo-token>
```

You can add this to your `.bashrc` script or similar shell profile to ensure it's always set.

Configure the repository

All the software needed for this example is available from the EDB Postgres Distributed package repository. Download and run a script to configure the EDB Postgres Distributed repository. This repository also contains the TPA packages.

```
curl -1sLf "https://downloads.enterprisedb.com/$EDB_SUBSCRIPTION_TOKEN/postgres_distributed/setup.deb.sh" | sudo
-E bash
```

Installing Trusted Postgres Architect (TPA)

You'll use TPA to provision and deploy PGD. If you previously installed TPA, you can move on to the next step. You'll find full instructions for installing TPA in the Trusted Postgres Architect documentation, which we've also included here.

Linux environment

TPA supports several distributions of Linux as a host platform. These examples are written for Ubuntu 22.04, but steps are similar for other supported platforms.

Install the TPA package

```
sudo apt install tpaexec
```

Configuring TPA

You now need to configure TPA, which configures TPA's Python environment. Call `tpaexec` with the command `setup`:

```
sudo /opt/EDB/TPA/bin/tpaexec setup
export PATH=$PATH:/opt/EDB/TPA/bin
```

You can add the `export` command to your shell's profile.

Testing the TPA installation

You can verify TPA is correctly installed by running `selftest`:

```
tpaexec selftest
```

TPA is now installed.

Installing PGD using TPA

Generating a configuration file

Run the `tpaexec configure` command to generate a configuration folder:

```
tpaexec configure democluster \
    --architecture PGD-Always-ON \
    --platform bare \
    --edb-postgres-advanced 16 \
    --redwood \
    --no-git \
    --location-names dc1 \
    --pgd-proxy-routing local \
    --hostnames-unsorted
```

You specify the PGD-Always-ON architecture (`--architecture PGD-Always-ON`), which sets up the configuration for PGD's Always-on architectures. As part of the default architecture, it configures your cluster with three data nodes, cohosting three PGD Proxy servers and a Barman node for backup.

For Linux hosts, specify that you're targeting a "bare" platform (`--platform bare`). TPA determines the Linux version running on each host during deployment. See the EDB Postgres Distributed compatibility table for details about the supported operating systems.

Specify that the data nodes will be running EDB Postgres Advanced Server v16 (`--edb-postgres-advanced 16`) with Oracle compatibility (`--redwood`).

You set the notional location of the nodes to `dc1` using `--location-names`. You then set `--pgd-proxy-routing` to `local` so that proxy routing can route traffic to all nodes in each location.

By default, TPA commits configuration changes to a Git repository. For this example, you don't need to do that, so pass the `--no-git` flag.

Finally, you ask TPA to generate repeatable hostnames for the nodes by passing `--hostnames-unsorted`. Otherwise, it selects hostnames at random from a predefined list of suitable words.

This command creates a subdirectory in the current working directory called `democluster`. It contains the `config.yml` configuration file TPA uses to create the cluster. You can view it using:

```
less democluster/config.yml
```

You now need to edit the configuration file to add details related to your Linux hosts, such as admin user names and public and private IP addresses.

Editing your configuration

Using your preferred editor, open `democluster/config.yml`.

Search for the line containing `ansible_user: root`. Change `root` to the name of the user you configured with SSH access and sudo privileges. Follow that with this line:

```
    manage_ssh_hostkeys: yes
```

Your `instance_defaults` section now looks like this:

```
instance_defaults:
  platform: bare
  vars:
    ansible_user: rocky
    manage_ssh_hostkeys: yes
```

Next, search for `node: 1`, which is the configuration settings of the first node, kaboom.

After the `node: 1` line, add the public and private IP addresses of your node. Use `linuxhost-1` as the host for this node. Add the following to the file, substituting your IP addresses. Align the start of each line with the start of the `node:` line.

```
   public_ip: 172.19.16.27
  private_ip: 192.168.2.247
```

The whole entry for kaboom looks like this but with your own IP addresses:

```
- Name:
kaboom
  backup: kapok
  location:
dc1
  node: 1
  public_ip: 172.19.16.27
  private_ip: 192.168.2.247
  role:
  -
bdr
  - pgd-proxy
  vars:
    bdr_child_group: dc1_subgroup
    bdr_node_options:
      route_priority: 100
```

Repeat this process for the three other nodes.

Search for `node: 2`, which is the configuration settings for the node kaftan. Use `linuxhost-2` for this node. Substituting your IP addresses, add:

```
  public_ip: 172.19.16.26
 private_ip: 192.168.2.41
```

Search for `node: 3`, which is the configuration settings for the node kaolin. Use `linuxhost-3` for this node. Substituting your IP addresses, add:

```
  public_ip: 172.19.16.25
 private_ip: 192.168.2.254
```

Finally, search for `node: 4`, which is the configuration settings for the node kapok. Use `linuxhost-4` for this node. Substituting your IP addresses, add:

```
  public_ip: 172.19.16.15
 private_ip: 192.168.2.30
```

Provisioning the cluster

You can now run:

```
tpaexec provision democluster
```

This command prepares for deploying the cluster. (On other platforms, such as Docker and AWS, this command also creates the required hosts. When using Linux hosts, your hosts must already be configured.)

> **Further reading**
>
> - `tpaexec provision` in the Trusted Postgres Architect documentation

One part of this process for Linux hosts is creating key-pairs for the hosts for SSH operations later. With those key-pairs created, you need to copy the public part of the key-pair to the hosts. You can do this with `ssh-copy-id`, giving the democluster identity (`-i`) and the login to each host. For this example, these are the commands:

```
ssh-copy-id -i democluster/id_democluster rocky@172.19.16.27
ssh-copy-id -i democluster/id_democluster rocky@172.19.16.26
ssh-copy-id -i democluster/id_democluster rocky@172.19.16.25
ssh-copy-id -i democluster/id_democluster rocky@172.19.16.15
```

You can now create the `tpa_known_hosts` file, which allows the hosts to be verified. Use `ssh-keyscan` on each host (`-H`) and append its output to `tpa_known_hosts`:

```
ssh-keyscan -H 172.19.16.27 >> democluster/tpa_known_hosts
ssh-keyscan -H 172.19.16.26 >> democluster/tpa_known_hosts
ssh-keyscan -H 172.19.16.25 >> democluster/tpa_known_hosts
ssh-keyscan -H 172.19.16.15 >> democluster/tpa_known_hosts
```

Deploy your cluster

You now have everything ready to deploy your cluster. To deploy, run:

```
tpaexec deploy democluster
```

TPA applies the configuration, installing the needed packages and setting up the actual EDB Postgres Distributed cluster.

> **Further reading**
>
> - `tpaexec deploy` in the Trusted Postgres Architect documentation

Connecting to the cluster

You're now ready to log in to one of the nodes of the cluster with SSH and then connect to the database. Part of the configuration process set up SSH logins for all the nodes, complete with keys. To use the SSH configuration, you need to be in the `democluster` directory created by the `tpaexec configure` command earlier:

```
cd democluster
```

From there, you can run `ssh -F ssh_config <hostname>` to establish an SSH connection. Connect to kaboom, the first database node in the cluster:

```
ssh -F ssh_config kaboom
```

output

```
[rocky@kaboom ~]#
```

Notice that you're logged in as rocky, the admin user and ansible user you configured earlier, on kaboom.

You now need to adopt the identity of the enterprisedb user. This user is preconfigured and authorized to connect to the cluster's nodes.

```
sudo -iu enterprisedb
```

output

```
enterprisedb@kaboom:~ $
```

You can now run the `psql` command to access the `bdrdb` database:

```
psql bdrdb
```

output

```
psql (16.2.0, server 16.2.0)
Type "help" for help.

bdrdb=#
```

You're directly connected to the Postgres database running on the kaboom node and can start issuing SQL commands.

To leave the SQL client, enter `exit` .

Using PGD CLI

The pgd utility, also known as the PGD CLI, lets you control and manage your EDB Postgres Distributed cluster. It's already installed on the node.

You can use it to check the cluster's health by running `pgd check-health` :

```
pgd check-health
```

output

```
 Check      Status Message
-----      ------ -------
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
enterprisedb@kaboom:~ $
```

Or, you can use `pgd show-nodes` to ask PGD to show you the data-bearing nodes in the cluster:

```
pgd show-nodes
```

output

```
 Node   Node ID    Group        Type Current State Target State Status Seq ID
----   -------    -----        ---- ------------- ------------ ------ ------
kaboom 2710197610 dc1_subgroup data ACTIVE        ACTIVE       Up     1
kaftan 3490219809 dc1_subgroup data ACTIVE        ACTIVE       Up     3
kaolin 2111777360 dc1_subgroup data ACTIVE        ACTIVE       Up     2
enterprisedb@kaboom:~ $
```

Similarly, use `pgd show-proxies` to display the proxy connection nodes:

```
pgd show-proxies
```

output

```
 Proxy  Group        Listen Addresses Listen Port
-----  -----        ---------------- -----------
kaboom dc1_subgroup [0.0.0.0]        6432
kaftan dc1_subgroup [0.0.0.0]        6432
kaolin dc1_subgroup [0.0.0.0]        6432
```

The proxies provide high-availability connections to the cluster of data nodes for applications. You can connect to the proxies and, in turn, to the database with the command `psql -h kaboom,kaftan,kaolin -p 6432 bdrdb`:

```
psql -h kaboom,kaftan,kaolin -p 6432 bdrdb
```

output

```
 psql (16.2.0, server 16.2.0)
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, compression: off)
Type "help" for help.

bdrdb=#
```

Explore your cluster

- Connect to your database to applications.
- Explore replication with hands-on exercises.
- Explore failover with hands-on exercises.
- Understand conflicts by creating and monitoring them.
- Take the next steps for working with your cluster.

6.3 Deploying an EDB Postgres Distributed example cluster on AWS

This quick start sets up EDB Postgres Distributed with an Always-on Single Location architecture using Amazon EC2.

Introducing TPA and PGD

We created TPA to make installing and managing various Postgres configurations easily repeatable. TPA orchestrates creating and deploying Postgres. In this quick start, you install TPA first. If you already have TPA installed, you can skip those steps. You can use TPA to deploy various configurations of Postgres clusters.

PGD is a multi-master replicating implementation of Postgres designed for high performance and availability. The installation of PGD is orchestrated by TPA. You'll use TPA to generate a configuration file for a PGD demonstration cluster. This cluster uses Amazon EC2 instances configures your cluster with three data nodes, cohosting three PGD Proxy servers, along with a Barman node for backup. You can then use TPA to provision and deploy the required configuration and software to each node.

Preparation

> **Note**
>
> This set of steps is specifically for Ubuntu 22.04 LTS on Intel/AMD processors.

EDB account

To install both TPA and PGD, you need an EDB account.

Sign up for a free EDB account if you don't already have one. Signing up gives you a trial subscription to EDB's software repositories.

After you're registered, go to the EDB Repos 2.0 page, where you can obtain your repo token.

On your first visit to this page, select **Request Access** to generate your repo token. Copy the token using the **Copy Token** icon, and store it safely.

Install curl

You use the `curl` command to retrieve installation scripts from repositories. On Ubuntu, curl isn't installed by default. To see if it's present, run `curl` in the terminal:

```
$ curl
Command 'curl' not found, but can be installed with:
sudo apt install curl
```

If not found, run:

```
sudo apt -y install curl
```

Setting environment variables

First, set the `EDB_SUBSCRIPTION_TOKEN` environment variable to the value of your EDB repo token, obtained in the EDB account step.

```
export EDB_SUBSCRIPTION_TOKEN=<your-repo-token>
```

You can add this to your `.bashrc` script or similar shell profile to ensure it's always set.

Configure the repository

All the software needed for this example is available from the EDB Postgres Distributed package repository. The following command downloads and runs a script to configure the EDB Postgres Distributed repository. This repository also contains the TPA packages.

```
curl -1sLf "https://downloads.enterprisedb.com/$EDB_SUBSCRIPTION_TOKEN/postgres_distributed/setup.deb.sh" | sudo
-E bash
```

> **Troubleshooting repo access**
>
> The script should produce output starting with:
>
> ```
> Executing the  setup script for the 'enterprisedb/postgres_distributed' repository ...
> ```
>
> If it produces no output or an error, double-check that you entered your token correctly. If the problem persists,contact Support for assistance.

Installing Trusted Postgres Architect (TPA)

You'll use TPA to provision and deploy PGD. If you previously installed TPA, you can move on to thenext step. You'll find full instructions for installing TPA in the Trusted Postgres Architect documentation, which we've also included here.

Linux environment

TPA supports several distributions of Linux as a host platform. These examples are written for Ubuntu 22.04, but steps are similar for other supported platforms.

Install the TPA package

```
sudo apt install tpaexec
```

Configuring TPA

You need to configure TPA, which configures TPA's Python environment. Call `tpaexec` with the command `setup`:

```
sudo /opt/EDB/TPA/bin/tpaexec setup
export PATH=$PATH:/opt/EDB/TPA/bin
```

You can add the `export` command to your shell's profile.

Testing the TPA installation

You can verify TPA is correctly installed by running `selftest`:

```
tpaexec selftest
```

TPA is now installed.

AWS Credentials

TPA uses your AWS credentials to perform the deployment onto AWS. Unless you have a corporate-managed account, you need to get your credentials from AWS. Corporate-managed accounts have their own process for obtaining credentials.

Your credentials consist of an AWS Access Key ID and a Secret Access Key. You also need to select an AWS default region for your work.

Set the environment variables `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`, and `AWS_DEFAULT_REGION` to the values of your AWS credentials. To ensure they're always set, you can add these to your `.bashrc` or similar shell profile.

```
$ export AWS_ACCESS_KEY_ID=THISISJUSTANEXAMPLE
$ export AWS_SECRET_ACCESS_KEY=d0ntU5E/Th1SAs1ts/jUs7anEXAMPLEKEY
$ export AWS_DEFAULT_REGION=us-west-2
```

Your account needs the necessary permissions to create and manage the resources that TPA uses. TPA AWS platform details the permissions that you need. Consult your AWS administrator if you need help with this.

Installing PGD using TPA

Generating a configuration file

Run the `tpaexec configure` command to generate a configuration folder:

```
tpaexec configure democluster \
  --architecture PGD-Always-ON \
  --platform aws \
  --region eu-west-1 \
  --edb-postgres-advanced 16 \
  --redwood \
  --location-names dc1 \
  --pgd-proxy-routing local \
  --no-git \
  --hostnames-unsorted
```

You specify the PGD-Always-ON architecture (`--architecture PGD-Always-ON`), which sets up the configuration for PGD's Always-on architectures. As part of the default architecture, this configures your cluster with three data nodes, cohosting three PGD Proxy servers, along with a Barman node for backup.

Specify that you're using AWS (`--platform aws`) and eu-west-1 as the region (`--region eu-west-1`).

TPA defaults to t3.micro instances on AWS. This is enough for this demonstration and also suitable for use with an AWS free tier account.

> **AWS free tier limitations**
>
> AWS free tier limitations for EC2 are based on hours of instance usage. Depending on how much time you spend testing, you might exceed these limits and incur charges.

By default, TPA configures Debian as the default OS for all nodes on AWS.

> **Deployment platforms**
>
> Other Linux platforms are supported as deployment targets for PGD. See the EDB Postgres Distributed compatibility table for details.
>
> Observe that you don't have to deploy PGD to the same platform you're using to run TPA!

Specify that the data nodes will be running EDB Postgres Advanced Server v16 (`--edb-postgres-advanced 16`) with Oracle compatibility (`--redwood`).

You set the notional location of the nodes to `dc1` using `--location-names`. You then set `--pgd-proxy-routing` to `local` so that proxy routing can route traffic to all nodes in each location.

By default, TPA commits configuration changes to a Git repository. For this example, you don't need to do that, so you pass the `--no-git` flag.

Finally, you ask TPA to generate repeatable hostnames for the nodes by passing `--hostnames-unsorted`. Otherwise, it selects hostnames at random from a predefined list of suitable words.

This command creates a subdirectory in the current working directory called `democluster`. It contains the `config.yml` configuration file TPA uses to create the cluster. You can view it using:

```
less democluster/config.yml
```

> **Further reading**
>
> - View the full set of available options by running:
>
>   ```
>   tpaexec configure --architecture PGD-Always-ON --help
>   ```
>
> - More details on PGD-Always-ON configuration options in Deploying with TPA
> - PGD-Always-ON in the TPA documentation
> - `tpaexec configure` in the TPA documentation
> - AWS platform in the TPA documentation

Provisioning the cluster:

Next, allocate the resources needed to run the configuration you just created using the `tpaexec provision` command:

```
tpaexec provision democluster
```

Since you specified AWS as the platform (the default platform), TPA provisions EC2 instances, VPCs, subnets, routing tables, internet gateways, security groups, EBS volumes, elastic IPs, and so on.

Because you didn't specify an existing one when configuring, TPA also prompts you to confirm the creation of an S3 bucket.

> **Remember to remove the bucket when you're done testing!**
>
> TPA doesn't remove the bucket that it creates in this step when you later deprovision the cluster. Take note of the name now, so that you can be sure to remove it later.

> **Further reading**
>
> - `tpaexec provision` in the Trusted Postgres Architect documentation

Deploying the cluster

With configuration in place and infrastructure provisioned, you can now deploy the distributed cluster:

```
tpaexec deploy democluster
```

TPA applies the configuration, installing the needed packages and setting up the actual EDB Postgres Distributed cluster.

> **Further reading**
>
> - `tpaexec deploy` in the Trusted Postgres Architect documentation

Connecting to the cluster

You're now ready to log in to one of the nodes of the cluster with SSH and then connect to the database. Part of the configuration process is to set up SSH logins for all the nodes, complete with keys. To use the SSH configuration, you need to be in the `democluster` directory created by the `tpaexec configure` command earlier:

```
cd democluster
```

From there, you can run `ssh -F ssh_config <hostname>` to establish an SSH connection. You will connect to kaboom, the first database node in the cluster:

```
ssh -F ssh_config kaboom
```

output

```
[admin@kaboom ~]#
```

Notice that you're logged in as admin on kaboom.

You now need to adopt the identity of the enterprisedb user. This user is preconfigured and authorized to connect to the cluster's nodes.

```
sudo -iu enterprisedb
```

output

```
enterprisedb@kaboom:~ $
```

You can now run the `psql` command to access the bdrdb database:

```
psql bdrdb
```

output

```
psql (16.2.0, server 16.2.0)
Type "help" for help.

bdrdb=#
```

You're directly connected to the Postgres database running on the kaboom node and can start issuing SQL commands.

To leave the SQL client, enter `exit`.

Using PGD CLI

The pgd utility, also known as the PGD CLI, lets you control and manage your EDB Postgres Distributed cluster. It's already installed on the node.

You can use it to check the cluster's health by running `pgd check-health`:

```
pgd check-health
```

```
output
Check      Status Message
-----      ------ -------
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
enterprisedb@kaboom:~ $
```

Or, you can use `pgd show-nodes` to ask PGD to show you the data-bearing nodes in the cluster:

```
pgd show-nodes
```

```
output
Node   Node ID    Group        Type Current State Target State Status Seq ID
----   -------    -----        ---- ------------- ------------ ------ ------
kaboom 2710197610 dc1_subgroup data ACTIVE        ACTIVE       Up     1
kaftan 3490219809 dc1_subgroup data ACTIVE        ACTIVE       Up     3
kaolin 2111777360 dc1_subgroup data ACTIVE        ACTIVE       Up     2
enterprisedb@kaboom:~ $
```

Similarly, use `pgd show-proxies` to display the proxy connection nodes:

```
pgd show-proxies
```

```
output
Proxy  Group        Listen Addresses Listen Port
-----  -----        ---------------- -----------
kaboom dc1_subgroup [0.0.0.0]        6432
kaftan dc1_subgroup [0.0.0.0]        6432
kaolin dc1_subgroup [0.0.0.0]        6432
```

The proxies provide high-availability connections to the cluster of data nodes for applications. You can connect to the proxies and, in turn, to the database with the command `psql -h kaboom,kaftan,kaolin -p 6432 bdrdb`:

```
psql -h kaboom,kaftan,kaolin -p 6432 bdrdb
```

```
output
psql (16.2.0, server 16.2.0)
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, compression: off)
Type "help" for help.

bdrdb=#
```

Explore your cluster

- Connect your database to applications.
- Explore replication with hands-on exercises.
- Explore failover with hands-on exercises.
- Understand conflicts by creating and monitoring them.

- Take the next steps for working with your cluster.

6.4 Deploying an EDB Postgres Distributed example cluster on Azure and Google cloud platforms

Deploying on Azure and Google clouds

For most cloud platforms, such as Azure and Google Cloud Platform, you create Linux hosts on the cloud platform you're using. You can then use the Deploying on Linux hosts quick start to deploy PGD to those Linux hosts.

- Azure users can follow a Microsoft guide on how to provision Azure VMs loaded with Linux.
- Google Cloud Platform users can follow a Google guide on how to provision GCP VMs with Linux loaded.

Then continue with Deploying on Linux hosts.

For AWS users, see the Deploying on AWS quick start. Using this quick start, TPA provisions the hosts needed to create your cluster.

6.5 Connecting to your database

Connecting your application or remotely connecting to your new EDB Postgres Distributed cluster involves:

- Getting credentials and optionally creating a `.pgpass` file
- Establishing the IP address of any PGD Proxy hosts you want to connect to
- Ensuring that you can connect to that IP address
- Getting an appropriate Postgres client
- Connecting the client to the cluster

Getting credentials

The default user, enterprisedb, was created in the cluster by tpaexec. It also generated passwords for that user as part of the provisioning. To get the password, run:

```
tpaexec show-password democluster enterprisedb
```

This command returns a string that's the password for the enterprisedb user. If you want, you can use that string when prompted for a password.

Creating a .pgpass file

You can avoid entering passwords for psql and other Postgres clients by creating a `.pgpass` file in your home directory. It contains password details that applications can look up when connecting. After getting the password (see Getting credentials), you can open the `.pgpass` file using your preferred editor.

In the file, enter:

```
*:*:bdrdb:enterprisedb:<your password>
```

Save the file and exit the editor. To secure the file, run the following command. This command gives read and write access only to you.

```
chmod 0600 ~/.pgpass
```

Establishing the IP address

Docker

Your Docker quick start cluster is by default accessible on the IP addresses 172.17.0.2 (kaboom), 172.17.0.3 (kaftan), 172.17.04 (kaolin), and 172.17.0.5 (kapok). Docker generates these addresses.

AWS

You can refer to the IP addresses in `democluster/ssh_config`. Alternatively, run:

```
aws ec2 --region eu-west-1 describe-instances --query 'Reservations[*].Instances[*].
{PublicIpAddress:PublicIpAddress,Name:Tags[?Key==`Name`]|[0].Value}'
```

output

```
[
    [
        {
            "PublicIpAddress": "54.217.130.13",
            "Name": "kapok"
        }
    ],
    [
        {
            "PublicIpAddress": "54.170.119.101",
            "Name": "kaolin"
        }
    ],
    [
        {
            "PublicIpAddress": "3.250.235.130",
            "Name": "kaftan"
        }
    ],
    [
        {
            "PublicIpAddress": "34.247.188.211",
            "Name": "kaboom"
        }
    ]
]
```

This command shows you EC2's list of public IP addresses for the cluster instances.

Linux hosts

You set IP addresses for your Linux servers when you configured the cluster in the quick start. Use those addresses.

Ensure you can connect to your IP addresses

Linux hosts and Docker

You don't need to perform any configuration to connect these.

AWS

AWS is configured to allow outside access only to its SSH endpoints. To allow Postgres clients to connect from outside the AWS cloud, you need to enable the transit of traffic on port 6432.

Get your own external IP address or the external IP address of the system you want to connect to the cluster. One way to do this is to run:

```
curl https://checkip.amazonaws.com
```

output

```
89.97.100.108
```

You also need the security groupid for your cluster. Run:

```
aws ec2 --region eu-west-1 describe-security-groups --filter Name=group-name,Values="*democluster*" | grep
GroupId
```

output

```
            "GroupId": "sg-072f996360ba20d5c",
```

Enter the correct region for your cluster, which you set when you configured it.

```
aws ec2 authorize-security-group-ingress --group-id <SECURITYGROUPID> --protocol tcp --port 6432 --cidr
<YOURIP>/32 --region eu-west-1
```

Again, make sure you put in the correct region for your cluster.

You can read more about this command in Add rules to your security group in the AWS CLI guide.

Getting an appropriate Postgres client

Unless you installed Postgres on your local system, you probably need to install a client application, such as psql, to connect to your database cluster.

On Ubuntu, for example, you can run:

```
sudo apt install postgresql-client
```

This command installs psql, along with some other tools but without installing the Postgres database locally.

Connecting the client to the cluster

After you install psql or a similar client, you can connect to the cluster. Run:

```
psql -h <ipaddressofnode> -p 6432 -U enterprisedb bdrdb
```

output

```
psql (16.2, server 16.2.0)
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256, compression: off)
Type "help" for help.

bdrdb=#
```

Use the `.pgpass` file with clients that support it, or use the host, port, user, password, and database name to connect with other applications.

Using proxy failover to connect the client to the cluster

By listing all the addresses of proxies as the host, you can ensure that the client will always failover and connect to the first available proxy in the event of a proxy failing.

```
psql -h <ipaddressofnode1>,<ipaddressofnode2>,<ipaddressofnode3> -U enterprisedb -p 6432 bdrdb
```

output

```
psql (16.2, server 16.2.0)
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256, compression: off)
Type "help" for help.

bdrdb=#
```

Creating a connection URL

Many applications use a connection URL to connect to the database. To create a connection URL, you need to assemble a string in the format:

```
postgresql://<user>@<ipaddressofnode1>:6432,<ipaddressofnode2>:6432,<ipaddressofnode3>:6432/bdrdb
```

This format of the string can be used with the `psql` command, so if your database nodes are on IP addresses 192.168.9.10, 192.168.10.10, and 192.168.10.11, you can use:

```
psql postgresql://enterprisedb@192.168.9.10:6432,192.168.10.10:6432,192.168.11.10:6432/bdrdb
```

You can also embed the password in the created URL. If you're using the enterprisedb user, and the password for the enterprisedb user is `notasecret`, then the URL looks like:

```
psql postgresql://enterprisedb:notasecret@192.168.9.10:6432,192.168.10.10:6432,192.168.11.10:6432/bdrdb
```

Actual passwords are more complex than that and can contain special characters. You need to urlencode the password to ensure that it doesn't trip up the shell, the command, or the driver you're using.

> **Passwords should not be embedded**
>
> While we have shown you how to embed a password, we recommend that you do not do this. The password is easily extracted from the URL and can easily be saved in insecure locations. Consider other ways of passing the password.

Making a Java connection URL

Finally, the URL you created is fine for many Postgres applications and clients, but those based on Java require one change to allow Java to invoke the appropriate driver. Precede the URL with `jdbc:` to make a Java compatible URL:

```
jdbc:postgresql://enterprisedb@192.168.9.10:6432,192.168.10.10:6432,192.168.11.10:6432/bdrdb
```

Moving on

You're now equipped to connect your applications to your cluster, with all the connection credentials and URLs you need.

6.6 Exploring replication with PGD

Explore replication with PGD

With the cluster up and running, it's useful to run some basic checks to see how effectively it's replicating.

The following example shows one quick way to do this, but you must ensure that any testing you perform is appropriate for your use case.

Preparation

Ensure your cluster is ready to perform replication. If you haven't installed a cluster yet, use one of the quick start guides to get going.

1. Log in to the database on the first host.
2. Run `select bdr.wait_slot_confirm_lsn(NULL, NULL);`.

When the query returns, the cluster is ready.

Create data

The simplest way to test that the cluster is replicating is to log in to a node, create a table, populate it, and see the data you populated appear on a second node. On the first node:

1. Create a table:

```
CREATE TABLE quicktest ( id SERIAL PRIMARY KEY, value INT );
```

2. Populate the table:

```
INSERT INTO quicktest (value) SELECT
random()*10000
FROM
generate_series(1,10000);
```

3. Monitor replication performance:

```
select * from bdr.node_replication_rates;
```

4. Get a sum of the value column (for checking):

```
select COUNT(*),SUM(value) from quicktest;
```

Check data

1. To confirm the data was successfully replicated, log in to a second node.
    1. Get a sum of the value column (for checking):

    ```
    select COUNT(*),SUM(value) from quicktest;
    ```

    2. Compare with the result from the first node.

2. Log in to a third node.
    1. Get a sum of the value column (for checking):

        ```
        select COUNT(*),SUM(value) from quicktest;
        ```

    2. Compare with the result from the first and second nodes.

Worked example

Preparation

The cluster in this example has three data nodes: kaboom, kaftan, and kaolin. The example uses kaboom as the first node. Log in to kaboom and then into kaboom's Postgres server:

```
cd democluster
ssh -F ssh_config kaboom
sudo -iu enterprisedb psql bdrdb
```

Ensure the cluster is ready

To ensure that the cluster is ready to go, run:

```
select bdr.wait_slot_confirm_lsn(NULL, NULL);
```

output

```
 wait_slot_confirm_lsn
-----------------------


(1 row)
```

If the cluster is busy initializing, this query waits and returns when the cluster is ready.

Create data

On the first node (kaboom), create a table

Run:

```
CREATE TABLE quicktest ( id SERIAL PRIMARY KEY, value INT );
```

output

```
CREATE TABLE
```

On kaboom, populate the table

This command generates a table of 10000 rows of random values:

```
INSERT INTO quicktest (value) SELECT random()*10000 FROM generate_series(1,10000);
```

output

```
INSERT 0 10000
```

On kaboom, monitor performance

As soon as possible, run the following command. It shows statistics about how quickly that data was replicated to the other two nodes.

```
select * from bdr.node_replication_rates;
```

output

```
 peer_node_id | target_name | sent_lsn   | replay_lsn | replay_lag | replay_lag_bytes | replay_lag_size |
apply_rate | catchup_interval
--------------+-------------+------------+------------+------------+------------------+-----------------+------------
-----------------
   3490219809 | kaftan      | 0/F57D120 | 0/F57D120  | 00:00:00   |                0 | 0 bytes         |       9158
| 00:00:00
   2111777360 | kaolin      | 0/F57D120 | 0/F57D120  | 00:00:00   |                0 | 0 bytes         |       9293
| 00:00:00
(2 rows)
```

The `replay_lag` values are 0, showing no lag. The LSN values are in sync, meaning the data is already replicated.

On kaboom get a checksum

Run:

```
select COUNT(*),SUM(value) from quicktest;
```

This command calculates a sum of the values from the generated data:

```
bdrdb=# select COUNT(*),SUM(value) from quicktest;
```

output

```
 count  |    sum
--------+-----------
 100000 | 498884606
(1 row)
```

Your sum will be different because the values in the table are random numbers, but the count will be 100000.

Check data

The second host is kaftan. In another window or session, log in to kaftan's Postgres server:

```
cd democluster
ssh -F ssh_config kaftan
sudo -iu enterprisedb psql bdrdb
```

On the second node (kaftan), get a checksum

Run:

```
select COUNT(*),SUM(value) from quicktest;
```

This command gets the second node's values for the generated data:

```
bdrdb=# select COUNT(*),SUM(value) from quicktest;
```

output

```
 count  |    sum
--------+-----------
 100000 | 498884606
(1 row)
```

Compare with the result from the first node (kaboom)

The values are identical.

You can repeat the process with the third node (kaolin), or generate new data on any node and see it replicate to the other nodes.

Log in to the third node (kaolin)

The third and last node is kaolin. In another window or session, log in to kaolin and then to kaolin's Postgres server:

```
cd democluster
ssh -F ssh_config kaolin
sudo -iu enterprisedb psql bdrdb
```

On kaolin, get a checksum

Run:

```
select COUNT(*),SUM(value) from quicktest;
```

This command gets kaolin's values for the generated data:

```
bdrdb=# select COUNT(*),SUM(value) from quicktest;
```

output

```
 count  |    sum
--------+-----------
 100000 | 498884606
(1 row)
```

Compare the results

Compare the result from the first and second nodes (kaboom and kaftan) with the result from kaolin. The values are identical on all three nodes.

6.7 Exploring failover handling with PGD

With a high-availability cluster, the ability to failover is crucial to the overall resilience of the cluster. When the lead data nodes stops working for whatever reason, applications need to be able to continue working with the database with little or no interruption. For PGD, that means directing applications to the new lead data node, which takes over automatically. This is where PGD Proxy is useful. It works with the cluster and directs traffic to the lead data node automatically.

In this exercise, you'll create an application that sends data to the database regularly. Then you'll first softly switch lead data node by requesting a change through the PGD CLI. And then you'll forcibly shut down a database instance and see how PGD handles that.

Your quick started configuration

This exploration assumes that you created your PGD cluster using the quick start for Docker, the quick start for AWS, or the quick start for Linux hosts.

At the end of each quick start, you'll have a cluster with four nodes and these roles:

Host name	Host role
kaboom	PGD data node and pgd-proxy co-host
kaftan	PGD data node and pgd-proxy co-host
kaolin	PGD data node and pgd-proxy co-host
kapok	Barman backup node

You'll use these hostnames throughout this exercise.

> **A best practice recommendation**
>
> This example is based on the quick start configuration. For speed and simplicity, it uses the Barman backup server in place of creating a bastion server. It also uses the Barman login to the Postgres cluster.
>
> In a production environment, we recommend that you create a separate bastion server to run the failover experiment from and that you create an appropriate Postgres user to log in to the cluster.

Installing xpanes

> **Xpanes optional**
>
> We recommend the xpanes utility for this exercise. It allows you to easily switch between multiple terminal sessions. If you prefer to use multiple terminals, tmux, or another terminal multiplexer, you can do so. Just make sure you can easily switch between multiple terminal sessions.

You'll use xpanes, a utility that allows you to quickly create multiple terminal sessions that you can easily switch between. It isn't installed by default, so you have to install it. For this exercise, you launch xpanes from the system where you ran tpaexec to configure your quick-start cluster.

If the system is running Ubuntu, run:

```
sudo apt install software-properties-common
sudo add-apt-repository ppa:greymd/tmux-xpanes
sudo apt update
sudo apt install tmux-xpanes
```

These are the installation instructions from the xpanes repository. If you aren't on Ubuntu, the repository also contains installation instructions for other systems.

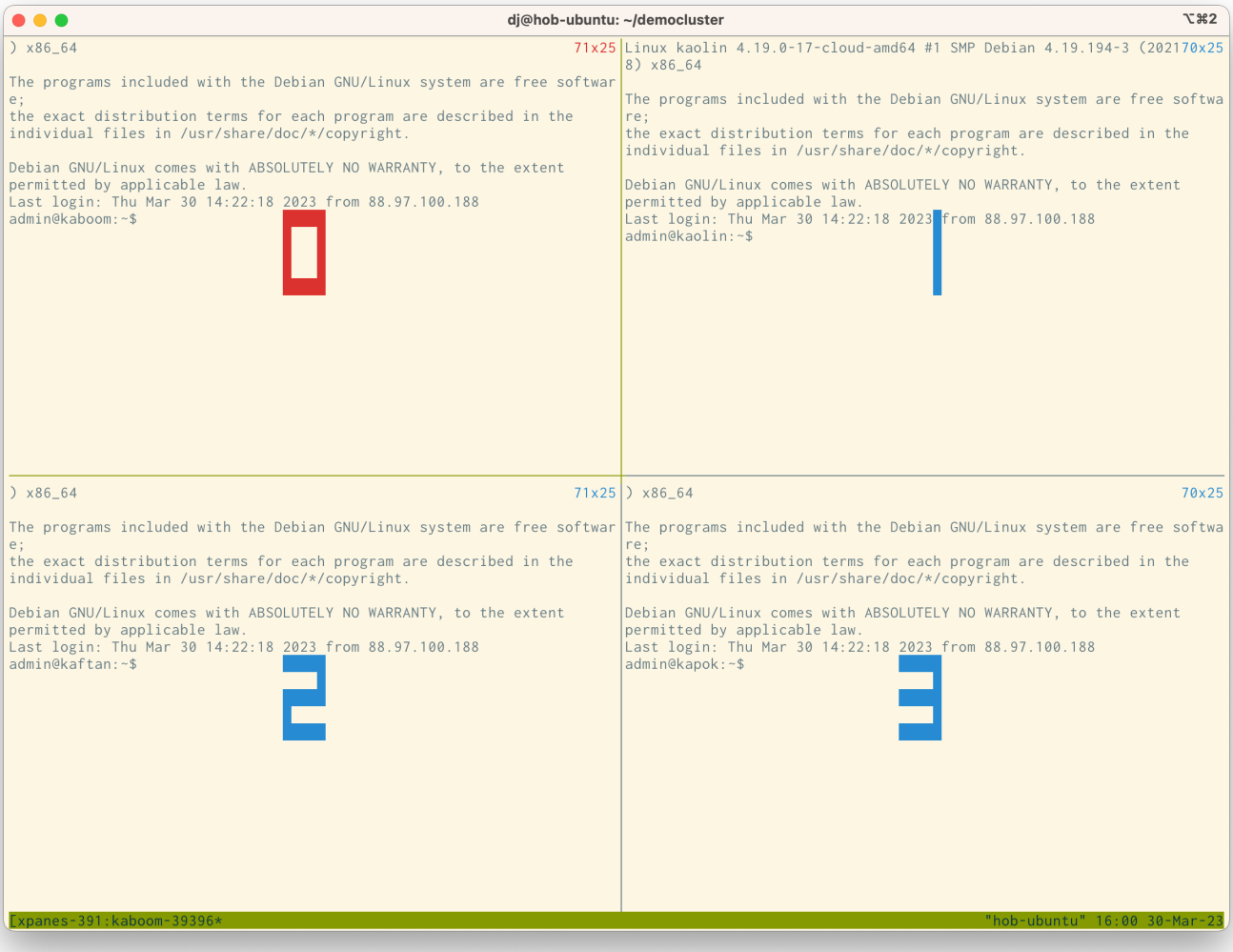
Connecting to the four servers

With xpanes installed, you can create an SSH session with all four servers by running:

```
cd democluster
xpanes -d -c "ssh -F ssh_config {}" "kaboom" "kaolin" "kaftan" "kapok"
```

After running these commands, there are four panes. The four panes are connected to kaboom, kaolin, kaftan, and kapok and you're logged in as the root user on each. You need this privilege so you can easily stop and start services later in the exercise.

Press **Control-b** followed by **q** to briefly display the numeric values for each pane.



To switch the focus between the panes, you can use **Control-b** and the cursor keys to navigate between them. Or you can use **Control-b** followed by **q** and the number of the pane you want to focus on. We'll show both ways.

Use **Control-b ↓ Control-b →** or **Control-b q 3** to move the focus to the bottom-right pane, which is the kapok host. This server is responsible for performing backups. You'll use this as the base of operations for your demo application. You can use Barman credentials to connect to the database servers and proxies:

```
sudo -iu barman
psql -h kaboom -p 6432 bdrdb
```

This code connects to the proxy on the kaboom host, which also runs a Postgres instance as part of the cluster.

The next step is to create the table for your application to write to:

```
drop table if exists ping cascade;
CREATE TABLE ping (id SERIAL PRIMARY KEY, node TEXT, timestamp TEXT) ;
```

This code first drops the `ping` table. Then it re-creates the `ping` table with an id primary key and two text fields for a node and timestamp. The table should now be ready. To verify that it is, use **Control-b ← Control-b ↑** or **Control-b q 0** to move to the top left pane, which puts you on the kaboom server. In this pane, become the enterprisedb user so you can easily connect to the database:

```
sudo -iu enterprisedb
```

You can now connect to the local database by running:

```
psql bdrdb
```

This command connects you directly to the local database instance on kaboom. Use `\dt` to view the available tables:

```
bdrdb=# \dt
       List of relations
 Schema | Name | Type  | Owner
--------+------+-------+--------
 public | ping | table | barman
(1 row)
```

Running `\d ping` shows that the DDL to create ping is on the kaboom server:

```
bdrdb=# \d ping
                          Table "public.ping"
  Column   |  Type   | Collation | Nullable | Default
-----------+---------+-----------+----------+----------------------------------
 id        | integer |           | not null | nextval('ping_id_seq'::regclass)
 node      | text    |           |          |
 timestamp | text    |           |          |
Indexes:
    "ping_pkey" PRIMARY KEY, btree (id)
```

If you want to be sure that this table is replicated, you can connect to another node in the cluster and look. The `\c` command in psql lets you connect to another server. To connect to the kaftan node, run:

```
\c - - kaftan
```

You'll see a login message similar to this:

```
psql.bin (16.2.0, server 16.2.0) SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, compression:
off)
You are now connected to database "bdrdb" as user "enterprisedb" on host "kaftan" (address "10.33.25.233") at
port "5444".
bdrdb=#
```

Run `\dt` and `\d ping`, and you'll see the same results on the kaftan node.

To reconnect to the kaboom node, run:

```
\c - - kaboom
```

Setting up a monitor

Next, you want to monitor the activity of the ping table. Enter this SQL to display the 10 most recent entries:

```
select * from ping order by timestamp desc limit 10;
```

To run this command more than once, use the `\watch` command in the shell, which executes the last query at regular intervals. To update every second, enter:

```
\watch 1
```

So far, there's nothing to see. You'll add activity soon.

Creating pings

Return to the Barman host kapok by using **Control-b ↓ Control-b →** or **Control-b q 3**.

This session is still logged into the psql session. Since you next want to run a shell script, you need to exit psql. Press **Control-d**.

The shell prompt now reads:

```
barman@kapok:~$
```

If it says `admin@kapok` or `root@kapok`, run `sudo -iu barman` to become the Barman user again.

The application you'll create is simple. It gets the node to write to and a timestamp for the ping. Then, as quickly as it can, it writes a new ping to the ping table.

In the shell, enter:

```
while true; do psql -h kaftan,kaolin,kaboom -p 6432 bdrdb -c "INSERT INTO ping(node, timestamp) select node_name,
current_timestamp from bdr.local_node_summary;"; done
```

In a more readable form, that is:

```
while true;
    do psql -h kaftan,kaolin,kaboom -p 6432 bdrdb -c \
        "INSERT INTO ping(node, timestamp) select node_name, current_timestamp from bdr.local_node_summary;"
    done
```

In a constant loop, you call the `psql` command, telling it to connect to any of the three proxies as hosts, giving the proxy port and selecting the bdrdb database. You also pass a command that inserts two values into the ping table. One of the values comes from `bdr.local_node_summary`, which contains the name of the node you're actually connected to. The other value is the current time.

Once the loop is running, new entries appear in the table. You'll see them in the top-left pane where you set up the monitor.

You can now start testing failover.

Displaying the write leader

For this part of the process, switch to the host kaftan, which is in the lower-left corner. Use**Control-b ←** or **Control-b q 2** to switch focus to it.

To gain appropriate privileges to run pgd, at the PGD command line interface, run:

```
sudo -iu enterprisedb
```

To see the state of the cluster, run:

```
pgd show-groups
```

You'll see output like this:

```
Group        Group ID   Type   Parent Group Location Raft Routing Write Leader
-----        --------   ----   ------------ -------- ---- ------- ------------
democluster  1935823863 global                       true false
dc1_subgroup 1302278103 data   democluster  dc1      true true    kaboom
```

The global group `democluster` includes all the subgroups. The `dc1_subgroup` is the data cluster you're working with. That group name value is derived from the location given in the quick start when you configured this cluster. Each location gets its own subgroup so you can manage it independently of other locations, or clusters.

If you skip to the right of the table, you can see that the current write leader for the group—the server where all the proxies send their updates—is kaboom.

Send a `switchover` command to the cluster group to change leader. Run this command:

```
pgd switchover --group-name dc1_subgroup --node-name kaolin
```

The node name is the host name for another data node in the dc1_subgroup group.

You'll see one of two responses. When you ran the `show-groups` command, if it showed kaolin as the write leader, you'll see:

```
Error: "kaolin" is already a write leader
```

This means that kaolin was already elected write leader, so switching has no effect. For this exercise, retry the switchover to another host, substituting kaboom or kaftan as the node name.

When you select a host that wasn't the current write leader, you'll see the other response:

```
switchover is complete
```

If you look in the top-left pane, you'll see the inserts from the script switching and being written to the node you just switched to.

> **Observe the id number**
>
> Notice that the id number being generated is from a completely different range of values, too. That's because the system transparently made the sequence generating the ID a global sequence. For more about global sequences and how they work, see Sequences.

You might also notice an error in the lower-right pane, as an inflight update is canceled by the switch. The script then continues writing.

Losing a node

Being able to switch leader is useful for planned maintenance; you tell the cluster to change configuration. What if unexpected changes happen? You'll create that scenario now.

In the lower-left pane, set the leader to kaolin.

```
pgd switchover --group-name dc1_subgroup --node-name kaolin
```

Then change focus to the top-right pane using **Control-b ↑ Control-b →** or **Control-b q 1**, which is the session on the kaolin host.

Turn off the Postgres server by running:

```
sudo systemctl stop postgres.service
```

In the top-left pane, you'll see the monitored table switch from kaolin to another node as the cluster subgroup picks a new leader. The script in the lower-right pane might show some errors as updates are canceled. However, as soon as a new leader is elected, it starts routing traffic to that leader.

Showing node states

Switch to the lower-left pane using **Control-b ↓ Control-b ←** or **Control-b q 2**, and run:

```
pgd show-nodes
```

You'll see something like:

```
 Node   Node ID    Group        Type Current State Target State Status      Seq ID
----   -------    -----        ---- ------------- ------------ ------      ------
kaboom 2710197610 dc1_subgroup data ACTIVE        ACTIVE       Up          3
kaftan 3490219809 dc1_subgroup data ACTIVE        ACTIVE       Up          2
kaolin 2111777360 dc1_subgroup data ACTIVE        ACTIVE       Unreachable 1
```

The kaolin node is down, and updates are going to a different write leader.

Monitoring lag

While kaolin is down, the logical replication at the heart of PGD is tracking how far out of sync kaolin is with the cluster. To see the details, run:

```
psql bdrdb -c "select * from bdr.node_replication_rates;"
```

This command displays the current replication rates between servers:

```
  peer_node_id | target_name | sent_lsn  | replay_lsn |   replay_lag    | replay_lag_bytes | replay_lag_size |
apply_rate | catchup_interval
--------------+-------------+-----------+------------+-----------------+------------------+-----------------+------
---+------------------
   2710197610 | kaboom      | 0/769F650 | 0/769F650  | 00:00:00        |                0 | 0 bytes         |
1861 | 00:00:00
              | kaolin      | 0/7656648 | 0/7656648  | 00:03:07.252266 |           299016 | 292 kB          |
|
(2 rows)
```

Looking at this output, you can see kaolin has a three-minute replay lag and around 292KB of data to catch up on if it came back now. The longer kaolin is down, the larger the replay lag gets. If you rerun the monitoring command, you'll see the numbers went up:

```
 peer_node_id | target_name | sent_lsn  | replay_lsn |   replay_lag    | replay_lag_bytes | replay_lag_size |
apply_rate | catchup_interval
--------------+-------------+-----------+------------+-----------------+------------------+-----------------+------
---+------------------
   2710197610 | kaboom      | 0/76B1D28 | 0/76B1D28  | 00:00:00        |                0 | 0 bytes         |
1743 | 00:00:00
              | kaolin      | 0/7656648 | 0/7656648  | 00:03:53.045704 |           374496 | 366 kB          |
|
(2 rows)
```

Another 46 seconds have passed, and the lag has grown by 74KB. Next, bring back the node, and see how the system recovers.

Restarting a node

You can bring back the Postgres service on kaolin. Switch back to the top-right pane using **Control-b ↑ Control-b →** or **Control-b q 1**, and run:

```
sudo systemctl start postgres.service
```

You won't see any change. Although the database service is back up and running, the cluster isn't holding an election, and so the leader remains in place. Switch to the lower-left pane using **Control-b ↓ Control-b ←** or **Control-b q 2**, and run:

```
pgd show-nodes
```

Now you'll see:

```
 Node   Node ID    Group        Type Current State Target State Status Seq ID
 ----   -------    -----        ---- ------------- ------------ ------ ------
 kaboom 2710197610 dc1_subgroup data ACTIVE        ACTIVE       Up     3
 kaftan 3490219809 dc1_subgroup data ACTIVE        ACTIVE       Up     2
 kaolin 2111777360 dc1_subgroup data ACTIVE        ACTIVE       Up     1
```

As soon as kaolin is back in the cluster, it begins synchronizing with the cluster. It does that by catching up on that replay data. Run:

```
psql bdrdb -c "select * from bdr.node_replication_rates;"
```

The output looks like this:

```
 peer_node_id | target_name | sent_lsn  | replay_lsn | replay_lag | replay_lag_bytes | replay_lag_size |
apply_rate | catchup_interval
--------------+-------------+-----------+------------+------------+------------------+-----------------+-----------
-----------------
   2710197610 | kaboom      | 0/8092938 | 0/8092938  | 00:00:00   |                0 | 0 bytes         |       2321
| 00:00:00
   2111777360 | kaolin      | 0/8092938 | 0/8092938  | 00:00:00   |                0 | 0 bytes         |     337426
| 00:00:00
(2 rows)
```

As you can see, there's no replay lag now, as kaolin has completely caught up.

With kaolin fully back in service, you can leave everything as it is. There's no need to change the server that's write leader. The failover mechanism is always ready to bring another server up to write leader when needed.

If you want, you can make kaolin leader again by running:

```
pgd switchover --group-name dc1_subgroup --node-name kaolin
```

This command returns kaolin to write lead. The application's updates will follow, as the proxies track the write leader.

Proxy failover

Proxies can also failover. To experience this, make sure your focus is still on the lower-left pane, and run:

```
pgd show-proxies
```

You'll see:

```
Proxy  Group        Listen Addresses Listen Port
-----  -----        ---------------- -----------
kaboom dc1_subgroup [0.0.0.0]        6432
kaftan dc1_subgroup [0.0.0.0]        6432
kaolin dc1_subgroup [0.0.0.0]        6432
```

Enter `exit` to exit the enterprisedb user and return to the admin/root shell. You can now stop the proxy service on this node by running:

```
systemctl stop pgd-proxy.service
```

A brief error appears in the lower-right window as the script switches to another proxy. The write leader doesn't change, though, so the switch of proxy doesn't show in the top-left pane where the monitor query is running.

Bring the proxy service on kaftan back by running:

```
systemctl start pgd-proxy.service
```

> **Exiting tmux**
>
> You can quickly exit tmux and all the associated sessions. First terminate any running processes, as they otherwise continue running after the session is killed. Press **Control-B** and then enter `:kill-session`. This approach is simpler than quitting each pane's session one at a time using **Control-D** or `exit`.

Other scenarios

This example uses the quick start configuration of three data nodes and one backup node. You can configure a cluster to have two data nodes and a witness node, which is less resilient to a node failing. Or you can configure five data nodes, which is much more resilient to a node failing. With this configuration, you can explore how failover works for your applications. For clusters with multiple locations, the same basic rules apply: taking a server down elects a new write leader that proxies now point to.

Further reading

- Read more about the management capabilities of the PGD CLI.
- Learn more about monitoring replication using SQL.

6.8 Exploring conflict handling with PGD

In a multi-master architecture like PGD, conflicts happen. PGD is built to handle them.

A conflict can occur when one database node has an update from an application to a row and another node has a different update to the same row. This type of conflict is called a *row-level conflict*. Conflicts aren't errors. Resolving them effectively is core to how Postgres Distributed maintains consistency.

The best way to handle conflicts is to prevent them! Use PGD's Always-on architecture with proxies to ensure that your applications write to the same server in the cluster. is When conflicts occur, though, it's useful to know how PGD resolves them, how you can control that resolution, and how you can find out that they're happening. Row insertion and row updates are two actions that can cause conflicts.

To see how it works, you need to open a command line view of all the servers.

Your quick start configuration

This exploration assumes that you created your PGD cluster using the quick start for Docker, the quick start for AWS, or the quick start for Linux hosts.

At the end of each quick start, you'll have a cluster with four nodes and these roles:

Host name	Host role
kaboom	PGD data node and pgd-proxy co-host
kaftan	PGD data node and pgd-proxy co-host
kaolin	PGD data node and pgd-proxy co-host
kapok	Barman backup node

You'll use these hostnames throughout this exercise.

Installing xpanes

> **Xpanes optional**
>
> We recommend the `xpanes` utility for this exercise. It allows you to easily switch between multiple terminal sessions. If you prefer to use multiple terminals, tmux or another terminal multiplexer, you can do so. Just make sure you can easily switch between multiple terminal sessions.

You'll use `xpanes`, a utility that allows you to quickly create multiple terminal sessions that you can easily switch between. It isn't installed by default, so you'll have to install it. Start by connecting to the kaboom node with ssh:

```
cd democluster && ssh -F ssh_config kaboom
```

If you're running the quick start on Docker, you'll be using Rocky Linux, a Red Hat derivative. To perform the xpanes install, run:

```
dnf -y install xpanes
```

If you're running the quick start on AWS, you'll be using Debian Linux. To perform the xpanes install, run:

```
wget https://github.com/greymd/tmux-xpanes/releases/download/v4.1.4/tmux-xpanes_v4.1.4.deb
sudo apt -y install ./tmux-xpanes*.deb
rm tmux-xpanes*.deb
```

Connecting to four servers

You need to be logged in as the enterprisedb user to allow authentication to work:

```
sudo -iu enterprisedb
```

Then, run the following command to connect to three database servers and a proxy server:

```
xpanes -d -c "psql postgresql://enterprisedb@{}/bdrdb?sslmode=require" "kaboom:5444" "kaftan:5444" "kaolin:5444"
"kaboom:6432"
```

xpanes takes the command after `-c` and uses the values in the arguments that follow to create a command to run. That means that, after you run it, there will be four panes. Three panes will be connected to the database nodes kaboom, kaftan, and kaolin on port 5444. One will be connected to the pgd-proxy running on kaboom on port 6432. Each one will be logged into the database as enterprisedb.

Press **Control-b** followed by **q** to briefly display the numeric values for each pane.

To switch the focus between the panes, you can use **Control-b** and the cursor keys to navigate between them. Or you can use **Control-b** followed by **q** and the number of the pane you want to focus on. We'll show both ways.

Move to the bottom-left pane using **Control-b ↓ Control-b → or **Control-b q 3**.

Preparing for conflicts

To make a conflict, you first need a simple table. In the pane that currently has focus, enter:

```
drop table if exists test_conflict;
create table test_conflict(
  id integer primary key ,
  value_1 text);
```

Monitoring conflicts

In the pane that currently has focus, enter:

```
select * from bdr.conflict_history_summary
\watch 1
```

The `select` command displays the conflict history for the cluster. The `\watch 1` command is a psql command that reruns the preceding command every second.

You are now ready to generate a conflict.

Creating a conflict

The most basic form of conflict is when an insert happens to a table on two different nodes and both have the same primary key. You can now create that scenario and observe it.

Move to the top-left pane using **Control-b ↑** **Control-b ←** or **Control-b q 0**. This pane is the kaboom node. Start a transaction there, and insert a row:

```
start transaction;
insert into test_conflict values (1, 'from kaboom');
```

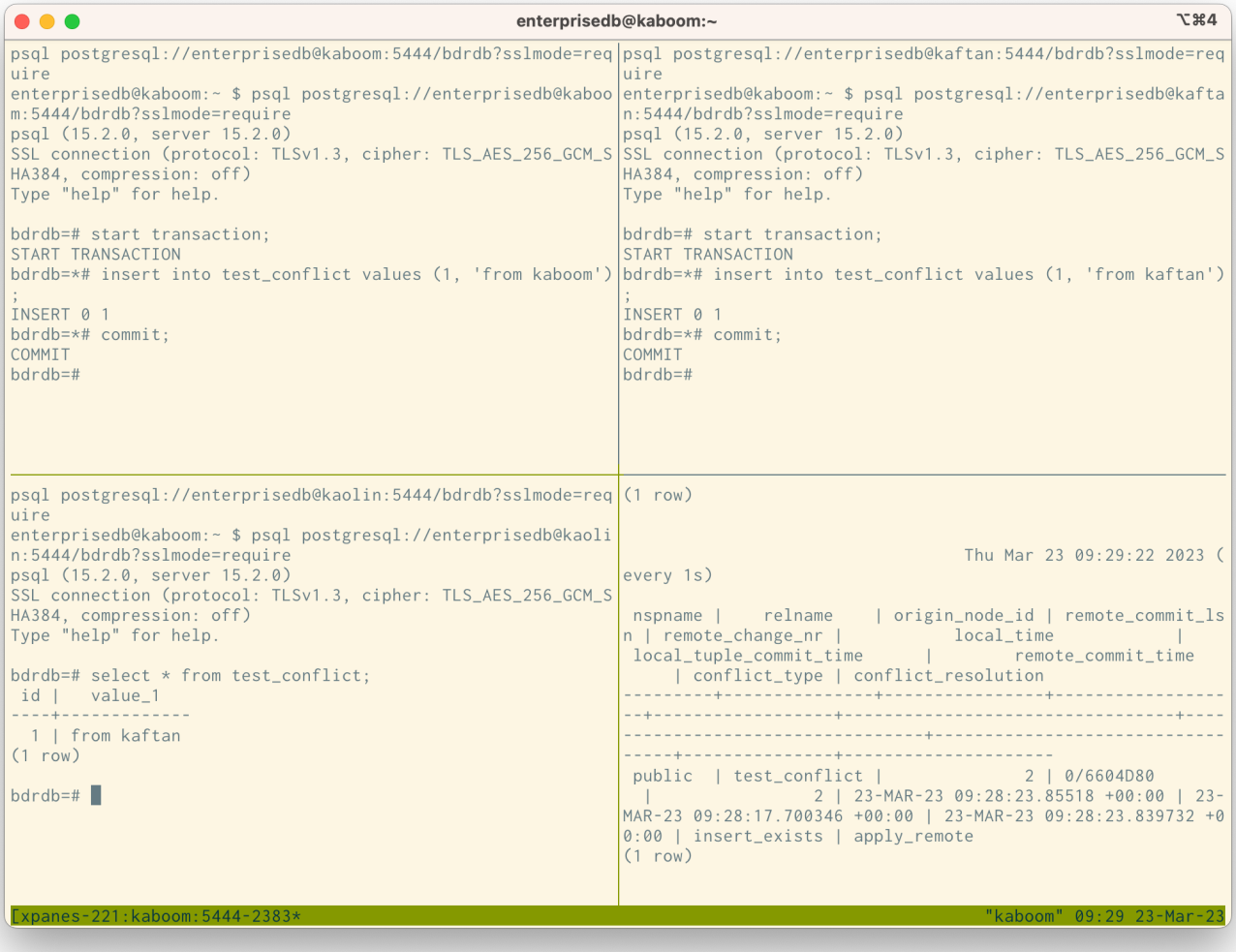
Next, move to the top-right pane using **Control-b →** or **Control-b q 1**. This pane is the kaftan node. Here, you'll also start a transaction and insert into the same row with different data:

```
start transaction;
insert into test_conflict values (1, 'from kaftan');
```

You now have two transactions open on different servers, with an insert operation already performed successfully. You need to commit both transactions at this point:

- Use **Control-b ←** or **Control-b q 0**, and then enter `commit;`.
- Use **Control-b →** or **Control-b q 1**, and then enter `commit;`.

You'll see that both commits are working. However, in the bottom-right pane, you can see the conflict being detected.



A row in the conflict history now notes a conflict in the table where the `insert_exists`. It also notes that the resolution for this conflict is that the newer record, based on the timing of the commit, is retained. This conflict is called an INSERT/INSERT conflict. You can read more about this type of conflict in INSERT/INSERT conflicts.

Creating an update conflict

When different updates to the same records take place on different nodes, a conflict occurs. You can create that scenario with the current configuration, too. Leave `\watch 1` running in the bottom-right pane.

Move to the top-left pane using **Control-b ←** or **Control-b q 0**. This pane is the kaboom node. Here, start a transaction and update a row:

```
start transaction;
update test_conflict set value_1 = 'update from kaboom' where id = 1;
```

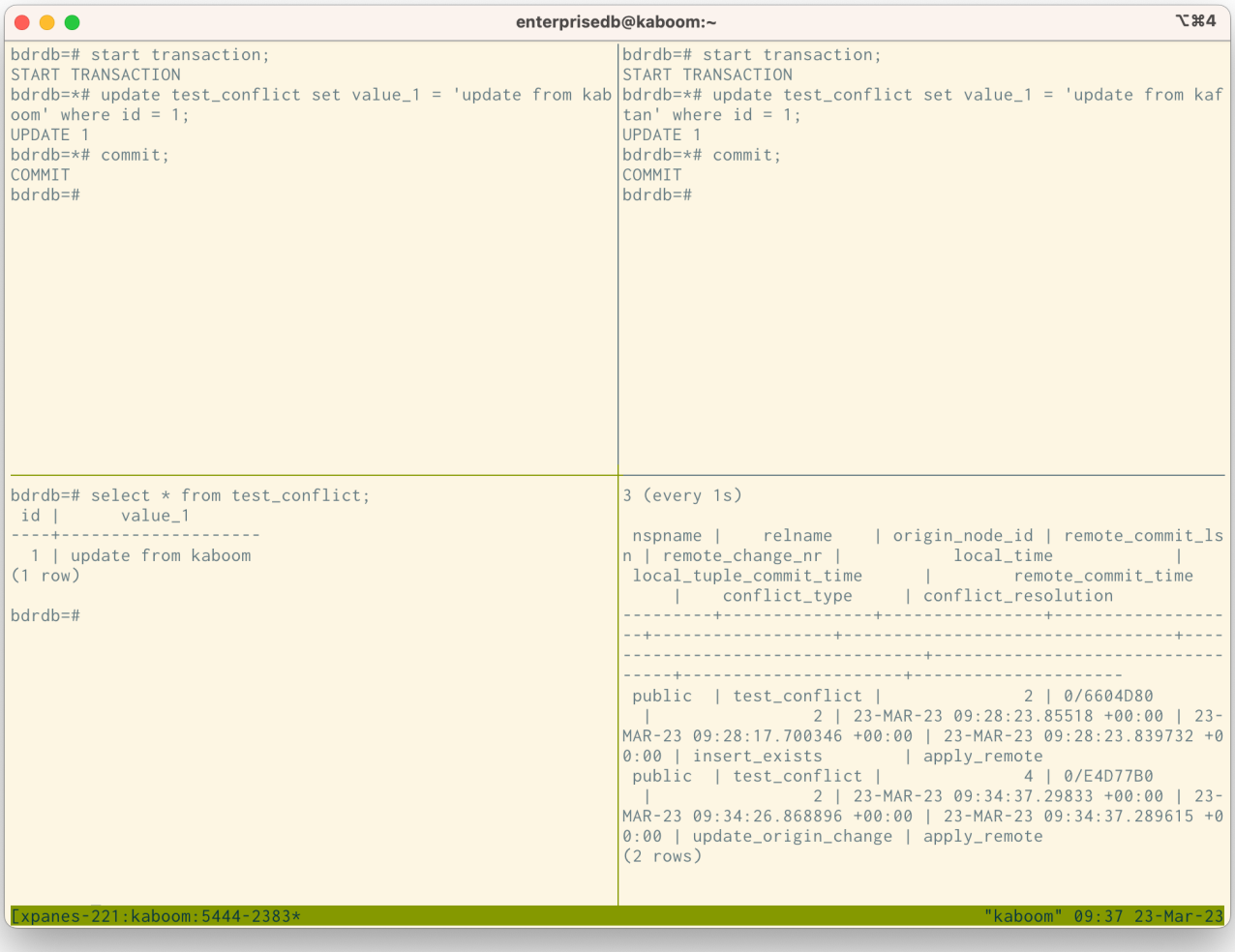
Next, move to the top-right pane using **Control-b →** or **Control-b q 1**. This pane is the kaftan node. Here, also start a transaction, and update the same row with different data:

```
start transaction;
update test_conflict set value_1 = 'update from kaftan' where id = 1;
```

You now have two transactions open on different servers, with an update operation already performed successfully. You need to commit both transactions at this point:

- Use **Control-b** ← or **Control-b q 0**, and then enter `commit;`.
- Use **Control-b** → or **Control-b q 1**, and then enter `commit;`.

Again you'll see both commits working. And, again, in the bottom-right pane, you can see the update conflict being detected.



An additional row in the conflict history shows an `update_origin_change` conflict occurred and that the resolution was `apply_remote`. This resolution means that the remote change was applied, updating the record. This conflict is called an UPDATE/UPDATE conflict and is explained in more detail in UPDATE/UPDATE conflicts.

> **Exiting tmux**
>
> You can quickly exit tmux and all the associated sessions. First terminate any running processes, as they will otherwise continue running after the session is killed. Press **Control-b** and then enter `:kill-session`. This approach is simpler than quitting each pane's session one at a time using **Control-D** or `exit`.

Other conflicts

You're now equipped to explore all the possible conflict scenarios and resolutions that can occur. For full details of how conflicts are managed, see Conflict management. While ideally you should avoid conflicts, it's important to know that, when they do happen, they're recorded and managed by Postgres Distributed's integrated and configurable conflict resolver.

6.9 Next steps with PGD

Going further with your PGD cluster

Architecture

This quick start created a single region cluster of high-availability Postgres databases. This is the Always-on Single Location architecture, one of a range of available PGD architectures. Other architectures include Always-on Multi-Location, with clusters in multiple data centers working together, and variations of both with witness nodes enhancing resilience. See architectural options.

Postgres versions

This quick start deployed EDB Postgres Advanced Server to the database nodes. PGD can deploy three different kinds of Postgres distributions, EDB Postgres Advanced Server, EDB Postgres Extended Server, and open-source PostgreSQL. The selection of database affects PGD, offering different capabilities, depending on server.

- Open-source PostgreSQL doesn't support CAMO.
- EDB Postgres Extended Server supports CAMO but doesn't offer Oracle compatibility.
- EDB Postgres Advanced Server supports CAMO and offers optional Oracle compatibility.

Further reading

- Learn PGD's terminology, from asynchronous replication to write scalability.
- Find out how applications work with PGD and how common Postgres features like sequences are globally distributed.
- Discover how PGD supports rolling upgrades of your clusters.
- Take control of routing and use SQL to control the PGD proxies.
- Engage with the PGD CLI to manage and monitor your cluster.

Deprovisioning the cluster

When you're done testing the cluster, deprovision it.

```
tpaexec deprovision democluster
```

- With a Docker deployment, deprovisioning tears down the Docker containers, network, and other local configuration.
- With an AWS deployment, deprovisioning removes the EC2 instances, VPC configuration, and other associated resources. Note that it leaves the S3 bucket it created. You must manually remove it.

7 Planning your PGD deployment

Planning your PGD deployment involves understanding the requirements of your application and the capabilities of PGD. This section provides an overview of the key considerations for planning your PGD deployment.

- Choosing your architecture: Understand the different architectures that PGD supports and choose the one that best fits your requirements.
- Choosing a Postgres distribution: Choose the Postgres distribution to deploy with PGD.
- Choosing your deployment method: Pick the deployment method that suits your needs.
- Other considerations: Consider other factors that may affect your deployment.
- Limitations: Know the limitations of PGD and their effect on your plans.

7.1 Choosing your architecture

Always-on architectures reflect EDB's Trusted Postgres architectures. They 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.

Always-on architectures can be deployed using EDB's standard deployment tool
Trusted Postgres Architect (TPA) or configured manually.

Standard EDB Always-on architectures

EDB has identified a set of standardized architectures to support single- or multi-location deployments with varying levels of redundancy, depending on your recovery point objective (RPO) and recovery time objective (RTO) requirements.

The Always-on architecture uses three database node groups as a basic building block. You can also use a five-node group for extra redundancy.

EDB Postgres Distributed consists of the following major building blocks:

- Bi-Directional Replication (BDR) — A Postgres extension that creates the multi-master mesh network
- PGD Proxy — A connection router that makes sure the application is connected to the right data nodes.

All Always-on architectures protect an increasing range of failure situations. For example, a single active location with two data nodes protects against local hardware failure but doesn't provide protection from location (data center or availability zone) failure. Extending that architecture with a backup at a different location ensures some protection in case of the catastrophic loss of a location. However, you still must restore the database from backup first, which might violate RTO requirements. Adding a second active location connected in a multi-master mesh network ensures that service remains available even if a location goes offline. Finally, adding a third location (this can be a witness-only location) allows global Raft functionality to work even if one location goes offline. The global Raft is primarily needed to run administrative commands. Also, some features like DDL or sequence allocation might not work without it, while DML replication can continue to work even without global Raft.

Each architecture can provide zero RPO, as data can be streamed synchronously to at least one local master, guaranteeing zero data loss in case of local hardware failure.

Increasing the availability guarantee always drives added cost for hardware and licenses, networking requirements, and operational complexity. It's important to carefully consider the availability and compliance requirements before choosing an architecture.

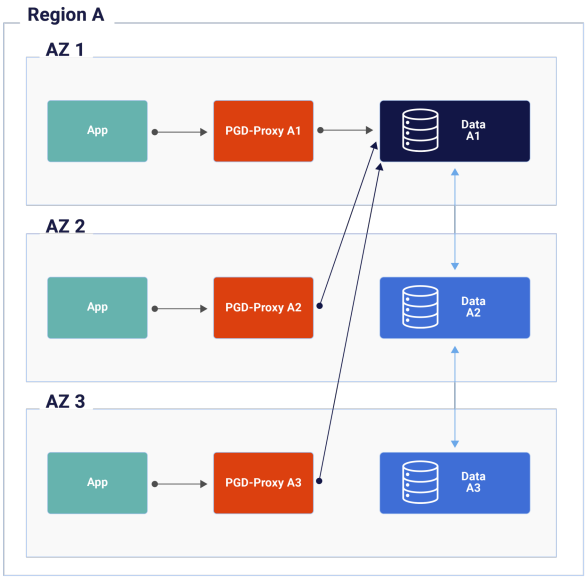
Architecture details

By default, application transactions don't require cluster-wide consensus for DML (selects, inserts, updates, and deletes), allowing for lower latency and better performance. However, for certain operations, such as generating new global sequences or performing distributed DDL, EDB Postgres Distributed requires an odd number of nodes to make decisions using a Raft-based consensus model. Thus, even the simpler architectures always have three nodes, even if not all of them are storing data.

Applications connect to the standard Always-on architectures by way of multi-host connection strings, where each PGD Proxy server is a distinct entry in the multi-host connection string. You must always have at least two proxy nodes in each location to ensure high availability. You can colocate the proxy with the database instance, in which case we recommend putting the proxy on every data node.

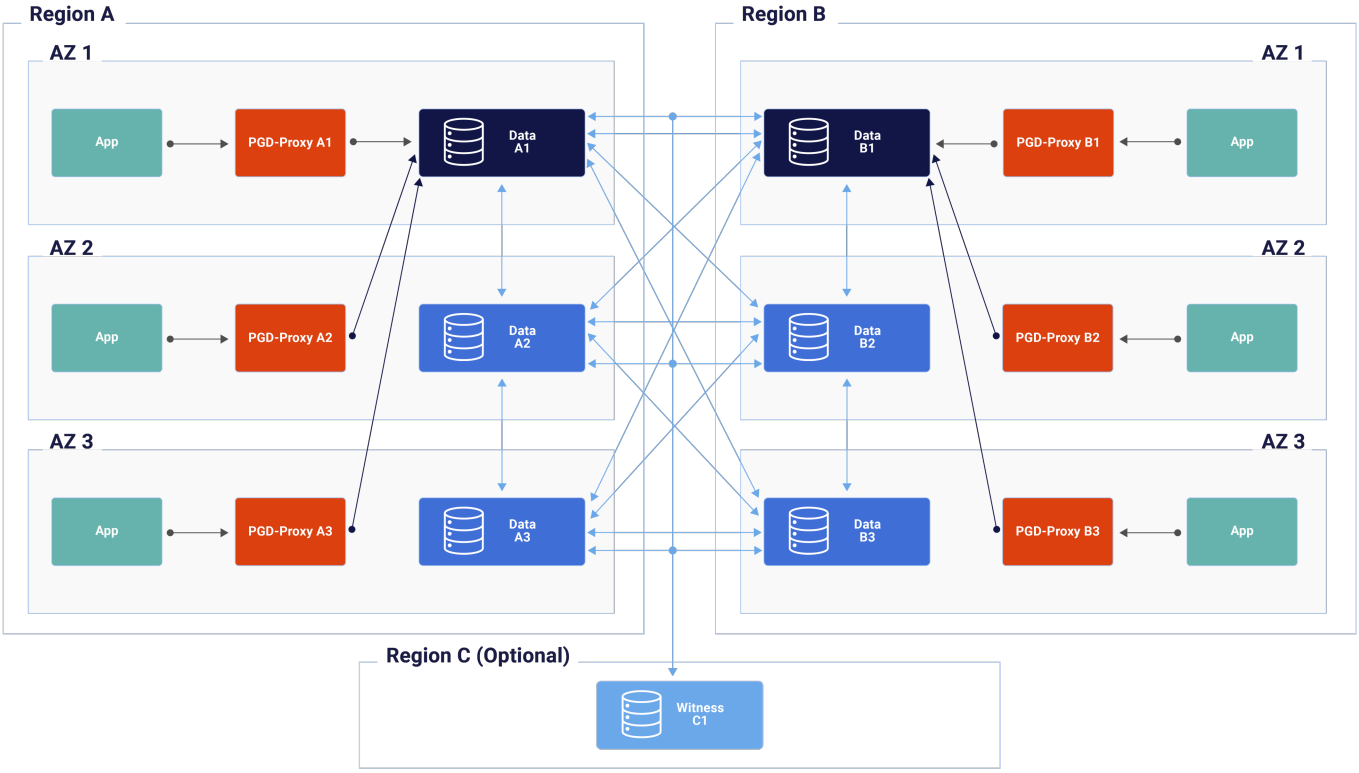
Other connection mechanisms have been successfully deployed in production. However, they aren't part of the standard Always-on architectures.

Always-on Single Location



- Additional replication between data nodes 1 and 3 isn't shown but occurs as part of the replication mesh
- Redundant hardware to quickly restore from local failures
  - 3 PGD nodes
    - Can be 3 data nodes (recommended)
    - Can be 2 data nodes and 1 witness that doesn't hold data (not depicted)
  - A PGD Proxy for each data node with affinity to the applications
    - Can be colocated with data node (recommended)
    - Can be located on a separate node
  - Configuration and infrastructure symmetry of data nodes is expected to ensure proper resources are available to handle application workload when rerouted
- Barman for backup and recovery (not depicted)
  - Offsite is optional but recommended
  - Can be shared by multiple PGD clusters
- Postgres Enterprise Manager (PEM) for monitoring (not depicted)
  - Can be shared by multiple PGD clusters

Always-on Multi-location



- Application can be Active/Active in each location or can be Active/Passive or Active DR with only one location taking writes.
- Additional replication between data nodes 1 and 3 isn't shown but occurs as part of the replication mesh.
- Redundant hardware to quickly restore from local failures.
    - 6 PGD nodes total, 3 in each location
        - Can be 3 data nodes (recommended)
        - Can be 2 data nodes and 1 witness which does not hold data (not depicted)
    - A PGD Proxy for each data node with affinity to the applications
        - Can be colocated with data node (recommended)
        - Can be located on a separate node
    - Configuration and infrastructure symmetry of data nodes and locations is expected to ensure proper resources are available to handle application workload when rerouted
- Barman for backup and recovery (not depicted).
    - Can be shared by multiple PGD clusters
- Postgres Enterprise Manager (PEM) for monitoring (not depicted).
    - Can be shared by multiple PGD clusters
- An optional witness node must be placed in a third region to increase tolerance for location failure.
    - Otherwise, when a location fails, actions requiring global consensus are blocked, such as adding new nodes and distributed DDL.

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.

	Single-data location	Two data locations	Two data locations + witness	Three or more data locations
Locations needed	1	2	3	3
Fast restoration of local HA after data node failure	Yes - if 3 PGD data nodes No - if 2 PGD data nodes	Yes - if 3 PGD data nodes No - if 2 PGD data nodes	Yes - if 3 PGD data nodes No - if 2 PGD data nodes	Yes - if 3 PGD data nodes No - if 2 PGD data nodes
Data protection in case of location failure	No (unless offsite backup)	Yes	Yes	Yes
Global consensus in case of location failure	N/A	No	Yes	Yes
Data restore required after location failure	Yes	No	No	No
Immediate failover in case of location failure	No - requires data restore from backup	Yes - alternate Location	Yes - alternate Location	Yes - alternate Location
Cross-location network traffic	Only if backup is offsite	Full replication traffic	Full replication traffic	Full replication traffic
License cost	2 or 3 PGD data nodes	4 or 6 PGD data nodes	4 or 6 PGD data nodes	6+ PGD data nodes

Adding flexibility to the standard architectures

To provide the data resiliency needed and proximity to applications and to the users maintaining the data, you can deploy the single-location architecture in as many locations as you want. While EDB Postgres Distributed has a variety of conflict-handling approaches available, do take care to minimize the number of expected collisions if allowing write activity from geographically disparate locations.

You can also expand the standard architectures with two additional types of nodes:

- *Subscriber-only nodes*, which you can use to achieve additional read scalability and to have data closer to users when the majority of an application's workload is read intensive with infrequent writes. You can also leverage them to publish a subset of the data for reporting, archiving, and analytic needs.

- *Logical standbys*, which receive replicated data from another node in the PGD cluster but don't participate in the replication mesh or consensus. They contain all the same data as the other PGD data nodes and can quickly be promoted to a master if one of the data nodes fails to return the cluster to full capacity/consensus. You can use them in environments where network traffic between data centers is a concern. Otherwise, three PGD data nodes per location is always preferred.

7.2 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 the Postgres distribution being used. Therefore, it's essential to adopt the Postgres distribution best suited to your business needs. For example, if having the Commit At Most Once (CAMO) feature is mission critical to your use case, don't adopt open source PostgreSQL, which doesn't 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
Rolling application and database upgrades	Y	Y	Y
Row-level last-update wins conflict resolution	Y	Y	Y
DDL replication	Y	Y	Y
Granular DDL Locking	Y	Y	Y
Streaming of large transactions	v14+	v13+	v14+
Distributed sequences	Y	Y	Y
Subscribe-only nodes	Y	Y	Y
Monitoring	Y	Y	Y
OpenTelemetry support	Y	Y	Y
Parallel apply	Y	Y	Y
Conflict-free replicated data types (CRDTs)	Y	Y	Y
Column-level conflict resolution	Y	Y	Y
Transform triggers	Y	Y	Y
Conflict triggers	Y	Y	Y
Asynchronous replication	Y	Y	Y
Legacy synchronous replication	Y	Y	Y
Group Commit	N	Y	14+
Commit At Most Once (CAMO)	N	Y	14+
Eager Conflict Resolution	N	Y	14+
Lag Control	N	Y	14+
Decoding Worker	N	13+	14+
Lag tracker	N	Y	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+
Transparent Data Encryption	N	15+	15+

7.3 Choosing your deployment method

You can deploy and install EDB Postgres Distributed products using the following methods:

- Trusted Postgres Architect (TPA) is an orchestration tool that uses Ansible to build Postgres clusters using 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 apply to quick testbed setups just as they do to production environments. TPA's flexibility allows deployments to virtual machines, AWS cloud instances, or Linux host hardware. See Deploying with TPA for more information.

- EDB Postgres AI Cloud Service is a fully managed database-as-a-service with built-in Oracle compatibility that runs in your cloud account or Cloud Service's cloud account where it's operated by EDB's Postgres experts. EDB Postgres AI Cloud Service makes it easy to set up, manage, and scale your databases. The addition of distributed high-availability support powered by EDB Postgres Distributed (PGD) enables single- and multi-region Always-on clusters. See Distributed high availability in the Cloud Service documentation for more information.

- EDB Postgres Distributed for Kubernetes is a Kubernetes operator designed, developed, and supported by EDB. It covers the full lifecycle of highly available Postgres database clusters with a multi-master architecture, using PGD replication. It's based on the open source CloudNativePG operator and provides additional value, such as compatibility with Oracle using EDB Postgres Advanced Server, Transparent Data Encryption (TDE) using EDB Postgres Extended or Advanced Server, and additional supported platforms including IBM Power and OpenShift.

	TPA	EDB Postgres AI Cloud Service	Kubernetes
Single region	✓	✓	✓
Active-Active support	2+ regions	2 regions	2 regions
Write/Read routing	Local or global	Local	Local
Automated failover	AZ or Region	AZ	AZ
Major version upgrades	✓	-	-
Subscriber-only nodes (read replicas)	✓	-	-
Logical standby nodes	✓	-	-
PgBouncer	✓	-	-
Selective data replication	✓	✓	✓
Maintenance windows per region	✓	✓	✓
Target availability	99.999% SLO	99.99 SLA (single) 99.995% SLA (multi)	99.999% SLO

7.4 Other considerations

Review these other considerations when planning your deployment.

Data consistency

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

Deployment

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

Log messages and documentation are currently available only in English.

Sizing considerations

For production deployments, EDB recommends a minimum of 4 cores for each Postgres data node. Witness nodes don't participate in the data replication operation and don't have to meet this requirement. One core is enough without subgroup Raft. Two cores are enough when using subgroup Raft. Always size logical standbys exactly like the data nodes to avoid performance degradations in case of a node promotion. In production deployments, PGD Proxy nodes require a minimum of 1 core and must increase incrementally with an increase in the number of database cores in approximately a 1:10 ratio. We recommend detailed benchmarking of your specific performance requirements to determine appropriate sizing based on your workload. The EDB Professional Services team is available to assist if needed.

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, Barman nodes, and PGD Proxy nodes on virtual machines or in a bare metal deployment mode. However, don't deploy multiple data nodes on VMs that are on the same physical hardware, as that reduces resiliency. Also don't deploy multiple PGD Proxy nodes on VMs on the same physical hardware, as that, too, reduces resiliency.

Single PGD Proxy nodes can be colocated with single PGD data nodes.

Clocks and timezones

EDB Postgres Distributed 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, though 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 affect 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.

7.5 Limitations

Take these EDB Postgres Distributed (PGD) design limitations into account when planning your deployment.

Nodes

- PGD can run hundreds of nodes, assuming adequate hardware and network. However, for mesh-based deployments, we generally don't recommend running more than 48 nodes in one cluster. If you need extra read scalability beyond the 48-node limit, you can add subscriber-only nodes without adding connections to the mesh network.
- The minimum recommended number of nodes in a group is three to provide fault tolerance for PGD's consensus mechanism. With just two nodes, consensus would fail if one of the nodes were unresponsive. Consensus is required for some PGD operations, such as distributed sequence generation. For more information about the consensus mechanism used by EDB Postgres Distributed, see Architectural details.

Multiple databases on single instances

Support for using PGD for multiple databases on the same Postgres instance is **deprecated** beginning with PGD 5 and will no longer be supported with PGD 6. As we extend the capabilities of the product, the added complexity introduced operationally and functionally is no longer viable in a multi-database design.

It's best practice and we recommend that you configure only one database per PGD instance.

The deployment automation with TPA and the tooling such as the CLI and PGD Proxy already codify that recommendation.

While it's still possible to host up to 10 databases in a single instance, doing so incurs many immediate risks and current limitations:

- If PGD configuration changes are needed, you must execute administrative commands for each database. Doing so increases the risk for potential inconsistencies and errors.
- You must monitor each database separately, adding overhead.
- TPAexec assumes one database. Additional coding is needed by customers or by the EDB Professional Services team in a post-deploy hook to set up replication for more databases.
- PGD Proxy works at the Postgres instance level, not at the database level, meaning the leader node is the same for all databases.
- Each additional database increases the resource requirements on the server. Each one needs its own set of worker processes maintaining replication, for example, logical workers, WAL senders, and WAL receivers. Each one also needs its own set of connections to other instances in the replication cluster. These needs might severely impact performance of all databases.
- Synchronous replication methods, for example, CAMO and Group Commit, won't work as expected. Since the Postgres WAL is shared between the databases, a synchronous commit confirmation can come from any database, not necessarily in the right order of commits.
- CLI and OTEL integration (new with v5) assumes one database.

Durability options (Group Commit/CAMO)

There are various limits on how the PGD durability options work. These limitations are a product of the interactions between Group Commit and CAMO, and how they interact with PGD features such as the WAL decoder and transaction streaming.

Also, there are limitations on interoperability with legacy synchronous replication, interoperability with explicit two-phase commit, and unsupported combinations within commit scope rules.

See Durability limitations for a full and current listing.

Mixed PGD versions

PGD was developed to enable rolling upgrades of PGD by allowing mixed versions of PGD to operate during the upgrade process. We expect users to run mixed versions only during upgrades and, once an upgrade starts, that they complete that upgrade. We don't support running mixed versions of PGD except during an upgrade.

Other limitations

This noncomprehensive list includes other limitations that are expected and are by design. We don't expect to resolve them in the future. Consider these limitations when planning your deployment:

- A `galloc` sequence might skip some chunks if you create the sequence in a rolled back transaction and then create it again with the same name. Skipping chunks can also occur if you create and drop the sequence when DDL replication isn't active and then you create it 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.

8 Deploying and configuring EDB Postgres Distributed

This section covers how to deploy EDB Postgres Distributed and how to configure it.

There are four main ways to deploy PGD:

- Manual deployment and administration describes how to manually deploy and configure EDB Postgres Distributed on a set of servers.
- Trusted Postgres Architect (TPA) describes how to use TPA to deploy and configure EDB Postgres Distributed to a Docker environment, Linux hosts, or AWS.
- EDB Postgres Distributed for Kubernetes describes how to deploy and configure EDB Postgres Distributed to a Kubernetes environment.
- EDB BigAnimal describes how to deploy and configure EDB Postgres Distributed on the EDB BigAnimal service.

8.1 Manual deployment and configuration of PGD

This section covers how to manually deploy and configure EDB Postgres Distributed.

- Deploying manually works through the steps needed to:
  - Provision hosts
  - Install Postgres
  - Configure repositories
  - Install the PGD software
  - Create a cluster
  - Check a cluster
  - Configure PGD proxies
  - Install and use PGD CLI

8.1.1 Deploying PGD manually

EDB offers automated PGD deployment using Trusted Postgres Architect (TPA) because it's generally more reliable than manual processes. SeeDeploying with TPA for full details about how to install TPA and use its automated best-practice-driven PGD deployment options. Or refer to any of the Quick start walkthroughs, which use TPA to get you up and running quickly.

To complement automated deployment, and to enable alternative installation and deployment processes, this section looks at the basic operations needed to manually configure a three-node PGD cluster (with a local subgroup), PGD Proxy, and PGD CLI.

Each step is outlined and followed by a worked example with further detail. This isn't a quick start guide but an exploration of PGD installation. It configures a basic deployment that will be used for further examples of PGD administration tasks.

The examples deploy a 3-node cluster of EDB Postgres Advanced Server 16 on Red Hat Enterprise Linux 9. These instructions also apply to RHEL derivatives like Alma Linux, Rocky Linux, or Oracle Linux.

At the highest level, manually deploying PGD involves the following steps. (For completeness, the steps also include instructions for installing PostgreSQL. If you're familiar with that, then skip to step 3.)

1: Provisioning and configuring hosts to run the cluster on

2: Installing your selected Postgres version on each of the hosts

3: Configuring repositories to install PGD to add PGD capabilities to each host

4: Initializing Postgres and installing PGD software on each host

5: Connecting the cluster by connecting to each node and telling it to join the cluster

6: Checking the cluster by running SQL commands and confirming replication has happened

7: Configuring proxies by creating a proxy config file

8: Using PGD CLI, including how to install PGD CLI and how to use it to inspect and manage your cluster

8.1.1.1 Step 1 - Provisioning hosts

Provisioning hosts

The first step in the process of deploying PGD is to provision and configure hosts.

You can deploy to virtual machine instances in the cloud with Linux installed, on-premises virtual machines with Linux installed, or on-premises physical hardware, also with Linux installed.

Whichever supported Linux operating system and whichever deployment platform you select, the result of provisioning a machine must be a Linux system that you can access using SSH with a user that has superuser, administrator, or sudo privileges.

Each machine provisioned must be able to make connections to any other machine you're provisioning for your cluster.

On cloud deployments, you can do this over the public network or over a VPC.

On-premises deployments must be able to connect over the local network.

> **Cloud provisioning guides**
>
> If you're new to cloud provisioning, these guides may provide assistance:
>
> | Vendor | Platform | Guide |
> |---|---|---|
> | Amazon | AWS | Tutorial: Get started with Amazon EC2 Linux instances |
> | Microsoft | Azure | Quickstart: Create a Linux virtual machine in the Azure portal |
> | Google | GCP | Create a Linux VM instance in Compute Engine |

Configuring hosts

Create an admin user

We recommend that you configure an admin user for each provisioned instance. The admin user must have superuser or sudo (to superuser) privileges. We also recommend that the admin user be configured for passwordless SSH access using certificates.

Ensure networking connectivity

With the admin user created, ensure that each machine can communicate with the other machines you're provisioning.

In particular, the PostgreSQL TCP/IP port (5444 for EDB Postgres Advanced Server, 5432 for EDB Postgres Extended and community PostgreSQL) must be open to all machines in the cluster. If you plan to deploy PGD Proxy, its port must be open to any applications that will connect to the cluster. Port 6432 is typically used for PGD Proxy.

Worked example

For this example, three hosts with Red Hat Enterprise Linux 9 were provisioned:

- host-one
- host-two
- host-three

Each is configured with an admin user named admin.

These hosts were configured in the cloud. As such, each host has both a public and private IP address.

Name	Public IP	Private IP
host-one	172.24.117.204	192.168.254.166
host-two	172.24.113.247	192.168.254.247
host-three	172.24.117.23	192.168.254.135

For the example cluster, `/etc/hosts` was also edited to use those private IP addresses:

```
 192.168.254.166 host-one
192.168.254.247 host-two
192.168.254.135 host-three
```

8.1.1.2 Step 2 - Installing Postgres

Installing Postgres

You need to install Postgres on all the hosts.

An EDB account is required to use the EDB Repos 2.0 page where you can get installation instructions. Select your platform and Postgres edition. You're presented with 2 steps of instructions. The first step covers how to configure the required package repository. The second step covers how to install the packages from that repository.

Run both steps.

Worked example

This example installs EDB Postgres Advanced Server 16 on Red Hat Enterprise Linux 9 (RHEL 9).

EDB account

You need an EDB account to install both Postgres and PGD.

Use your EDB account to sign in to the EDB Repos 2.0 page where you can select your platform. Then scroll down the list to select the Postgres version you want to install:

- EDB Postgres Advanced Server
- EDB Postgres Extended
- PostgreSQL

When you select the version of the Postgres server you want, two steps are displayed.

1: Configuring repositories

For step 1, you can choose to use the automated script or step through the manual install instructions that are displayed. Your EDB repository token is inserted into these scripts by the EDB Repos 2.0 site. In the examples, it's shown as `XXXXXXXXXXXXXXXX`.

On each provisioned host, you either run the automatic repository installation script or use the manual installation steps. The automatic script looks like this:

```
curl -1sLf 'https://downloads.enterprisedb.com/XXXXXXXXXXXXXXXX/enterprise/setup.rpm.sh' | sudo -E bash
```

The manual installation steps look like this:

```
dnf install yum-utils
rpm --import 'https://downloads.enterprisedb.com/XXXXXXXXXXXXXXXX/enterprise/gpg.E71EB0829F1EF813.key'
curl -1sLf 'https://downloads.enterprisedb.com/XXXXXXXXXXXXXXXX/enterprise/config.rpm.txt?distro=el&codename=9' >
/tmp/enterprise.repo
dnf config-manager --add-repo '/tmp/enterprise.repo'
dnf -q makecache -y --disablerepo='*' --enablerepo='enterprisedb-enterprise'
```

2: Install Postgres

For step 2, run the command to install the packages:

```
sudo dnf -y install edb-as16-server
```

8.1.1.3 Step 3 - Configuring PGD repositories

Configuring PGD repositories

To install and run PGD requires that you configure repositories so that the system can download and install the appropriate packages.

Perform the following operations on each host. For the purposes of this exercise, each host is a standard data node, but the procedure would be the same for other node types, such as witness or subscriber-only nodes.

- Use your EDB account.
    - Obtain your EDB repository token from the EDB Repos 2.0 page.
- Set environment variables.
    - Set the `EDB_SUBSCRIPTION_TOKEN` environment variable to the repository token:

      ```
      export EDB_SUBSCRIPTION_TOKEN=<your-repo-token>
      ```
- Configure the repository.
    - Run the automated installer to install the repositories:

      > **Red Hat**
      >
      > ```
      > curl -1sLf
      > "https://downloads.enterprisedb.com/$EDB_SUBSCRIPTION_TOKEN/postgres_distributed/setup.rpm.sh" |
      > sudo -E bash
      > ```

      > **Ubuntu/Debian**
      >
      > ```
      > curl -1sLf
      > "https://downloads.enterprisedb.com/$EDB_SUBSCRIPTION_TOKEN/postgres_distributed/setup.deb.sh" |
      > sudo -E bash
      > ```

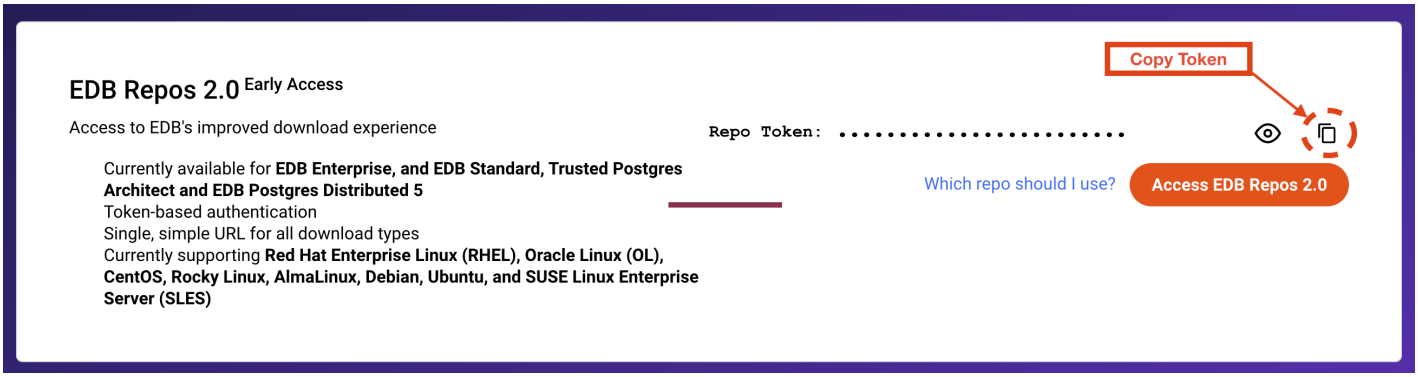
Worked example

Use your EDB account

You need an EDB account to install Postgres Distributed.

Use your EDB account to sign in to the EDB Repos 2.0 page, where you can obtain your repo token.

On your first visit to this page, select **Request Access** to generate your repo token.



Select **Copy Token** to copy the token to your clipboard, and store the token safely.

Set environment variables

Set the `EDB_SUBSCRIPTION_TOKEN` environment variable to the value of your EDB repo token, obtained in the EDB account step.

```
export EDB_SUBSCRIPTION_TOKEN=<your-repo-token>
```

You can add this to your `.bashrc` script or similar shell profile to ensure it's always set.

> **Note**
>
> Your preferred platform may support storing this variable as a secret, which can appear as an environment variable. If this is the case, add it to your platform's secret manager, and don't add the setting to `.bashrc`.

Configure the repository

All the software you need is available from the EDB Postgres Distributed package repository. You have the option to download and run a script to configure the EDB Postgres Distributed repository. You can also download, inspect, and then run that same script.

The following instructions also include the essential steps that the scripts take for any user wanting to manually run the installation process or to automate it.

RHEL/Other RHEL-based

You can autoinstall with automated OS detection:

```
curl -1sLf "https://downloads.enterprisedb.com/$EDB_SUBSCRIPTION_TOKEN/postgres_distributed/setup.rpm.sh" | sudo -E bash
```

If you want to inspect the script that's generated for you, run:

```
curl -1sLfO "https://downloads.enterprisedb.com/$EDB_SUBSCRIPTION_TOKEN/postgres_distributed/setup.rpm.sh"
```

Then inspect the resulting `setup.rpm.sh` file. When you're ready to proceed, run:

```
sudo -E bash setup.rpm.sh
```

If you want to perform all steps manually or use your own preferred deployment mechanism, you can use the following example as a guide.

You will need to pass details of your Linux distribution and version. You may need to change the codename to match the version of RHEL you're using. This example sets it for RHEL-compatible Linux version 9:

```
export DISTRO="el"
export CODENAME="9"
```

Now install the yum-utils package:

```
sudo dnf install -y yum-utils
```

The next step imports a GPG key for the repositories:

```
sudo rpm --import
"https://downloads.enterprisedb.com/$EDB_SUBSCRIPTION_TOKEN/postgres_distributed/gpg.B09F406230DA0084.key"
```

Now you can import the repository details, add them to the local configuration, and enable the repository.

```
curl -1sLf "https://downloads.enterprisedb.com/$EDB_SUBSCRIPTION_TOKEN/postgres_distributed/config.rpm.txt?
distro=$DISTRO&codename=$CODENAME" > /tmp/enterprise.repo
sudo dnf config-manager --add-repo '/tmp/enterprise.repo'
sudo dnf -q makecache -y --disablerepo='*' --enablerepo='enterprisedb-postgres_distributed'
```

8.1.1.4 Step 4 - Installing the PGD software

Installing the PGD software

With the repositories configured, you can now install the Postgres Distributed software. You must perform these steps on each host before proceeding to the next step.

- **Install the packages.**
  - Install the PGD packages, which include a server-specific BDR package and generic PGD Proxy and CLI packages. (`edb-bdr5-<postgresversion>` , `edb-pgd5-proxy` , and `edb-pgd5-cli`)
- **Ensure the Postgres database server has been initialized and started.**
  - Use `systemctl status` to check that the service is running.
  - If the service isn't running, initialize the database and start the service.
- **Configure the BDR extension.**
  - Add the BDR extension (`$libdir/bdr`) at the start of the `shared_preload_libraries` setting in `postgresql.conf` .
  - Set the `wal_level` GUC variable to `logical` in `postgresql.conf` .
  - Turn on commit timestamp tracking by setting `track_commit_timestamp` to `'on'` in `postgresql.conf` .
  - Increase the maximum worker processes to 16 or higher by setting `max_worker_processes` to `'16'` in `postgresql.conf` .

    > **The `max_worker_processes` value**
    >
    > The `max_worker_processes` value is derived from the topology of the cluster, the number of peers, number of databases, and other factors. To calculate the needed value, see Postgres configuration/settings. The value of 16 was calculated for the size of cluster being deployed in this example. It must be increased for larger clusters.

  - Set a password on the EnterprisedDB/Postgres user.
  - Add rules to `pg_hba.conf` to allow nodes to connect to each other.
    - Ensure that these lines are present in `pg_hba.conf` :

```
host all all all md5
host replication all all md5
```

  - Add a `.pgpass` file to allow nodes to authenticate each other.
    - Configure a user with sufficient privileges to log in to the other nodes.
    - See The Password File in the Postgres documentation for more on the `.pgpass` file.
- **Restart the server.**
  - Verify the restarted server is running with the modified settings and the BDR extension is available.
- **Create the replicated database.**
  - Log in to the server's default database (`edb` for EDB Postgres Advanced Server, `postgres` for PGE and community Postgres).
  - Use `CREATE DATABASE bdrdb` to create the default PGD replicated database.
  - Log out and then log back in to `bdrdb` .
  - Use `CREATE EXTENSION bdr` to enable the BDR extension and PGD to run on that database.

The worked example that follows shows the steps for EDB Postgres Advanced Server in detail.

If you're installing PGD with EDB Postgres Extended Server or community Postgres, the steps are similar, but details such as package names and paths are different. These differences are summarized in Installing PGD for EDB Postgres Extended Server and Installing PGD for Postgresql.

Worked example

Install the packages

The first step is to install the packages. Each Postgres package has an `edb-bdr5-<postgresversion>` package to go with it. For example, if you're installing EDB Postgres Advanced Server (epas) version 16, you'd install `edb-bdr5-epas16`.

There are two other packages to also install:

- `edb-pgd5-proxy` for PGD Proxy
- `edb-pgd5-cli` for the PGD command line tool

To install all of these packages on a RHEL or RHEL compatible Linux, run:

```
sudo dnf -y install edb-bdr5-epas16 edb-pgd5-proxy edb-pgd5-cli
```

Ensure the database is initialized and started

If the server wasn't initialized and started by the database's package initialization (or you're repeating the process), you need to initialize and start the server.

To see if the server is running, you can check the service. The service name for EDB Advanced Server is `edb-as-16`, so run:

```
sudo systemctl status edb-as-16
```

If the server isn't running, the response is:

```
○ edb-as-16.service - EDB Postgres Advanced Server 16
     Loaded: loaded (/usr/lib/systemd/system/edb-as-16.service; disabled; preset: disabled)
     Active: inactive (dead)
```

`Active: inactive (dead)` tells you that you need to initialize and start the server.

You need to know the path to the setup script for your particular Postgres flavor.

For EDB Postgres Advanced Server, you can find this script in `/usr/edb/as16/bin` as `edb-as-16-setup`. Run this command with the `initdb` parameter and pass an option to set the database to use UTF-8:

```
sudo PGSETUP_INITDB_OPTIONS="-E UTF-8" /usr/edb/as16/bin/edb-as-16-setup initdb
```

Once the database is initialized, start it so that you can continue configuring the BDR extension:

```
sudo systemctl start edb-as-16
```

Configure the BDR extension

Installing EDB Postgres Advanced Server creates a system user enterprisedb with admin capabilities when connected to the database. You'll use this user to configure the BDR extension.

Preload the BDR library

You need to preload the BDR library with other libraries. EDB Postgres Advanced Server has a number of libraries already preloaded, so you have to prefix the existing list with the BDR library.

```
echo -e "shared_preload_libraries = '\$libdir/bdr,\$libdir/dbms_pipe,\$libdir/edb_gen,\$libdir/dbms_aq'" | sudo -u enterprisedb tee -a /var/lib/edb/as16/data/postgresql.conf >/dev/null
```

> **Tip**
>
> This command format (`echo ... | sudo ... tee -a ...`) appends the echoed string to the end of the `postgresql.conf` file, which is owned by another user.

Set the `wal_level`

The BDR extension needs to set the server to perform logical replication. Do this by setting `wal_level` to `logical` :

```
echo -e "wal_level = 'logical'" | sudo -u enterprisedb tee -a /var/lib/edb/as16/data/postgresql.conf >/dev/null
```

Enable commit timestamp tracking

The BDR extension also needs the commit timestamp tracking enabled:

```
echo -e "track_commit_timestamp = 'on'" | sudo -u enterprisedb tee -a /var/lib/edb/as16/data/postgresql.conf >/dev/null
```

Increase `max_worker_processes`

To communicate between multiple nodes, Postgres Distributed nodes run more worker processes than usual. The default limit (8) is too low even for a small cluster.

The `max_worker_processes` value is derived from the topology of the cluster, the number of peers, number of databases, and other factors. To calculate the needed value, see Postgres configuration/settings.

This example, with a 3-node cluster, uses the value of 16.

Increase the maximum number of worker processes to 16:

```
echo -e "max_worker_processes = '16'" | sudo -u enterprisedb tee -a /var/lib/edb/as16/data/postgresql.conf >/dev/null
```

This value must be increased for larger clusters.

Add a password to the Postgres enterprisedb user

To allow connections between nodes, a password needs to be set on the Postgres enterprisedb user. This example uses the password `secret` . Select a different password for your deployments. You will need this password to Enable authentication between nodes.

```
sudo -u enterprisedb psql edb -c "ALTER USER enterprisedb WITH PASSWORD 'secret'"
```

Enable inter-node authentication in pg_hba.conf

Out of the box, Postgres allows local authentication and connections with the database but not external network connections. To enable this, edit `pg_hba.conf` and add appropriate rules, including rules for the replication users. To simplify the process, use this command:

```
echo -e "host all all all md5\nhost replication all all md5" | sudo tee -a /var/lib/edb/as16/data/pg_hba.conf
```

The command appends the following to `pg_hba.conf`:

```
host all all all md5
host replication all all md5
```

These commands enable the nodes to replicate.

Enable authentication between nodes

As part of the process of connecting nodes for replication, PGD logs into other nodes. It performs that login as the user that Postgres is running under. For EDB Postgres Advanced server, this is the enterprisedb user. That user needs credentials to log into the other nodes. Supply these credentials using the `.pgpass` file, which needs to reside in the user's home directory. The home directory for `enterprisedb` is `/var/lib/edb`.

Run this command to create the file:

```
echo -e "*:*:*:enterprisedb:secret" | sudo -u enterprisedb tee /var/lib/edb/.pgpass; sudo chmod 0600
/var/lib/edb/.pgpass
```

You can read more about the `.pgpass` file in The Password File in the PostgreSQL documentation.

Restart the server

After all these configuration changes, we recommend that you restart the server with:

```
sudo systemctl restart edb-as-16
```

Check the extension has been installed

At this point, it's worth checking whether the extension is actually available and the configuration was correctly loaded. You can query the `pg_available_extensions` table for the BDR extension like this:

```
sudo -u enterprisedb psql edb -c "select * from pg_available_extensions where name like 'bdr'"
```

This command returns an entry for the extension and its version:

```
 name | default_version | installed_version |                  comment
------+-----------------+-------------------+-------------------------------------------
 bdr  | 5.3.0           |                   | Bi-Directional Replication for PostgreSQL
```

You can also confirm the other server settings using this command:

```
sudo -u enterprisedb psql edb -c "show all" | grep -e wal_level -e track_commit_timestamp -e max_worker_processes
```

Create the replicated database

The server is now prepared for PGD. You need to next create a database named `bdrdb` and install the BDR extension when logged into it:

```
sudo -u enterprisedb psql edb -c "CREATE DATABASE bdrdb"
sudo -u enterprisedb psql bdrdb -c "CREATE EXTENSION bdr"
```

Finally, test the connection by logging in to the server.

```
sudo -u enterprisedb psql bdrdb
```

You're connected to the server. Execute the command "\dx" to list extensions installed:

```
bdrdb=# \dx
                              List of installed extensions
       Name       | Version |   Schema   |                   Description
------------------+---------+------------+-------------------------------------------------
 bdr              | 5.3.0   | pg_catalog | Bi-Directional Replication for PostgreSQL
 edb_dblink_libpq | 1.0     | pg_catalog | EnterpriseDB Foreign Data Wrapper for PostgreSQL
 edb_dblink_oci   | 1.0     | pg_catalog | EnterpriseDB Foreign Data Wrapper for Oracle
 edbspl           | 1.0     | pg_catalog | EDB-SPL procedural language
 plpgsql          | 1.0     | pg_catalog | PL/pgSQL procedural language
(5 rows)
```

Notice that the BDR extension is listed in the table, showing that it's installed.

Summaries

Installing PGD for EDB Postgres Advanced Server

For your convenience, here's a summary of the commands used in this example.

```
sudo dnf -y install edb-bdr5-epas16 edb-pgd5-proxy edb-pgd5-cli
sudo PGSETUP_INITDB_OPTIONS="-E UTF-8" /usr/edb/as16/bin/edb-as-16-setup initdb
sudo systemctl start edb-as-16
echo -e "shared_preload_libraries = '\$libdir/bdr,\$libdir/dbms_pipe,\$libdir/edb_gen,\$libdir/dbms_aq'" | sudo -
u enterprisedb tee -a /var/lib/edb/as16/data/postgresql.conf >/dev/null
echo -e "wal_level = 'logical'" | sudo -u enterprisedb tee -a /var/lib/edb/as16/data/postgresql.conf >/dev/null
echo -e "track_commit_timestamp = 'on'" | sudo -u enterprisedb tee -a /var/lib/edb/as16/data/postgresql.conf
>/dev/null
echo -e "max_worker_processes = '16'" | sudo -u enterprisedb tee -a /var/lib/edb/as16/data/postgresql.conf
>/dev/null
sudo -u enterprisedb psql edb -c "ALTER USER enterprisedb WITH PASSWORD 'secret'"
echo -e "host all all all md5\nhost replication all all md5" | sudo tee -a /var/lib/edb/as16/data/pg_hba.conf
>/dev/null
echo -e "*:*:*:enterprisedb:secret" | sudo -u enterprisedb tee /var/lib/edb/.pgpass >/dev/null; sudo chmod 0600
/var/lib/edb/.pgpass
sudo systemctl restart edb-as-16
sudo -u enterprisedb psql edb -c "CREATE DATABASE bdrdb"
sudo -u enterprisedb psql bdrdb -c "CREATE EXTENSION bdr"
sudo -u enterprisedb psql bdrdb
```

Installing PGD for EDB Postgres Extended Server

Installing PGD with EDB Postgres Extended Server has a number of differences from the EDB Postgres Advanced Server installation:

- The BDR package to install is named `edb-bdrV-pgextendedNN` (where V is the PGD version and NN is the PGE version number).
- Call a different setup utility: `/usr/edb/pgeNN/bin/edb-pge-NN-setup`.
- The service name is `edb-pge-NN`.
- The system user is postgres (not enterprisedb).
- The home directory for the postgres user is `/var/lib/pgqsl`.
- There are no preexisting libraries to add to `shared_preload_libraries`.

Summary: Installing PGD for EDB Postgres Extended Server 16

```
sudo dnf -y install edb-bdr5-pgextended16 edb-pgd5-proxy edb-pgd5-cli
sudo PGSETUP_INITDB_OPTIONS="-E UTF-8" /usr/edb/pge16/bin/edb-pge-16-setup ekend initdb
sudo systemctl start edb-pge-16
echo -e "shared_preload_libraries = '\$libdir/bdr'" | sudo -u postgres tee -a /var/lib/edb-
pge/16/data/postgresql.conf >/dev/null
echo -e "wal_level = 'logical'" | sudo -u postgres tee -a /var/lib/edb-pge/16/data/postgresql.conf >/dev/null
echo -e "track_commit_timestamp = 'on'" | sudo -u postgres tee -a /var/lib/edb-pge/16/data/postgresql.conf
>/dev/null
echo -e "max_worker_processes = '16'" | sudo -u postgres tee -a /var/lib/edb-pge/16/data/postgresql.conf
>/dev/null
sudo -u postgres psql postgres -c "ALTER USER postgres WITH PASSWORD 'secret'"
echo -e "host all all all md5\nhost replication all all md5" | sudo tee -a /var/lib/edb-pge/16/data/pg_hba.conf
>/dev/null
echo -e "*:*:*:postgres:secret" | sudo -u postgres tee /var/lib/pgsql/.pgpass >/dev/null; sudo chmod 0600
/var/lib/pgsql/.pgpass
sudo systemctl restart edb-pge-16
sudo -u postgres psql postgres -c "CREATE DATABASE bdrdb"
sudo -u postgres psql bdrdb -c "CREATE EXTENSION bdr"
sudo -u postgres psql bdrdb
```

Installing PGD for Postgresql

Installing PGD with PostgresSQL has a number of differences from the EDB Postgres Advanced Server installation:

- The BDR package to install is named `edb-bdrV-pgNN` (where V is the PGD version and NN is the PostgreSQL version number).
- Call a different setup utility: `/usr/pgsql-NN/bin/postgresql-NN-setup`.
- The service name is `postgresql-NN`.
- The system user is postgres (not enterprisedb).
- The home directory for the postgres user is `/var/lib/pgqsl`.
- There are no preexisting libraries to add to `shared_preload_libraries`.

Summary: Installing PGD for Postgresql 16

```
 sudo dnf -y install edb-bdr5-pg16 edb-pgd5-proxy edb-pgd5-cli
sudo PGSETUP_INITDB_OPTIONS="-E UTF-8" /usr/pgsql-16/bin/postgresql-16-setup initdb
sudo systemctl start postgresql-16
echo -e "shared_preload_libraries = '\$libdir/bdr'" | sudo -u postgres tee -a
/var/lib/pgsql/16/data/postgresql.conf >/dev/null
echo -e "wal_level = 'logical'" | sudo -u postgres tee -a /var/lib/pgsql/16/data/postgresql.conf >/dev/null
echo -e "track_commit_timestamp = 'on'" | sudo -u postgres tee -a /var/lib/pgsql/16/data/postgresql.conf
>/dev/null
echo -e "max_worker_processes = '16'" | sudo -u postgres tee -a /var/lib/pgsql/16/data/postgresql.conf >/dev/null
sudo -u postgres psql postgres -c "ALTER USER postgres WITH PASSWORD 'secret'"
echo -e "host all all all md5\nhost replication all all md5" | sudo tee -a /var/lib/pgsql/16/data/pg_hba.conf
>/dev/null
echo -e "*:*:*:postgres:secret" | sudo -u postgres tee /var/lib/pgsql/.pgpass; sudo chmod 0600
/var/lib/pgsql/.pgpass
sudo systemctl restart postgresql-16
sudo -u postgres psql postgres -c "CREATE DATABASE bdrdb"
sudo -u postgres psql bdrdb -c "CREATE EXTENSION bdr"
sudo -u postgres psql bdrdb
```

8.1.1.5 Step 5 - Creating the PGD cluster

Creating the PGD cluster

- **Create connection strings for each node.**
  For each node, create a connection string that will allow PGD to perform replication.

  The connection string is a key/value string that starts with a `host=` and the IP address of the host. (If you have resolvable named hosts, the name of the host is used instead of the IP address.)

  That's followed by the name of the database. In this case, use `dbname=bdrdb`, as a `bdrdb` database was created when installing the software.

  We recommend you also add the port number of the server to your connection string as `port=5444` for EDB Postgres Advanced Server and `port=5432` for EDB Postgres Extended and community PostgreSQL.

- **Prepare the first node.** To create the cluster, select and log in to the `bdrdb` database on any host's Postgres server.

- **Create the first node.**
  Run `bdr.create_node` and give the node a name and its connection string where *other* nodes can connect to it.

  - Create the top-level group. Create a top-level group for the cluster with `bdr.create_node_group`, giving it a single parameter: the name of the top-level group.
  - Create a subgroup. Create a subgroup as a child of the top-level group with `bdr.create_node_group`, giving it two parameters: the name of the subgroup and the name of the parent (and top-level) group. This process initializes the first node.

- **Add the second node.**

  - Create the second node. Log in to another initialized node's `bdrdb` database. Run `bdr.create_node` and give the node a different name and its connection string where *other* nodes can connect to it.
  - Join the second node to the cluster. Next, run `bdr.join_node_group`, passing two parameters: the connection string for the first node and the name of the subgroup you want the node to join.

- **Add the third node.**

  - Create the third node. Log in to another initialized node's `bdrdb` database. Run `bdr.create_node` and give the node a different name and its connection string where *other* nodes can connect to it.
  - Join the third node to the cluster. Next, run `bdr.join_node_group`, passing two parameters: the connection string for the first node and the name of the subgroup you want the node to join.

Worked example

So far, this example has:

- Created three hosts.
- Installed a Postgres server on each host.
- Installed Postgres Distributed on each host.
- Configured the Postgres server to work with PGD on each host.

To create the cluster, you tell host-one's Postgres instance that it's a PGD node—node-one—and create PGD groups on that node. Then you tell host-two and host-three's Postgres instances that they are PGD nodes—node-two and node-three—and that they must join a group on node-one.

Create connection strings for each node

Calculate the connection strings for each of the nodes in advance. Following are the connection strings for this 3-node example.

Name	Node name	Private IP	Connection string
host-one	node-one	192.168.254.166	host=host-one dbname=bdrdb port=5444
host-two	node-two	192.168.254.247	host=host-two dbname=bdrdb port=5444
host-three	node-three	192.167.254.135	host=host-three dbname=bdrdb port=5444

Preparing the first node

Log in to host-one's Postgres server.

```
ssh admin@host-one
sudo -iu enterprisedb psql bdrdb
```

Create the first node

Call the `bdr.create_node` function to create a node, passing it the node name and a connection string that other nodes can use to connect to it.

```
select bdr.create_node('node-one','host=host-one dbname=bdrdb port=5444');
```

Create the top-level group

Call the `bdr.create_node_group` function to create a top-level group for your PGD cluster. Passing a single string parameter creates the top-level group with that name. This example creates a top-level group named `pgd`.

```
select bdr.create_node_group('pgd');
```

Create a subgroup

Using subgroups to organize your nodes is preferred, as it allows services like PGD Proxy, which you'll configure later, to coordinate their operations. In a larger PGD installation, multiple subgroups can exist. These subgroups provide organizational grouping that enables geographical mapping of clusters and localized resilience. For that reason, this example creates a subgroup for the first nodes to enable simpler expansion and the use of PGD Proxy.

Call the `bdr.create_node_group` function again to create a subgroup of the top-level group. The subgroup name is the first parameter, and the parent group is the second parameter. This example creates a subgroup `dc1` as a child of `pgd`.

```
select bdr.create_node_group('dc1','pgd');
```

Adding the second node

Log in to host-two's Postgres server

```
ssh admin@host-two
sudo -iu enterprisedb psql bdrdb
```

Create the second node

Call the `bdr.create_node` function to create this node, passing it the node name and a connection string that other nodes can use to connect to it.

```
select bdr.create_node('node-two','host=host-two dbname=bdrdb port=5444');
```

Join the second node to the cluster

Using `bdr.join_node_group`, you can ask node-two to join node-one's `dc1` group. The function takes as a first parameter the connection string of a node already in the group and the group name as a second parameter.

```
select bdr.join_node_group('host=host-one dbname=bdrdb port=5444','dc1');
```

Add the third node

Log in to host-three's Postgres server.

```
ssh admin@host-three
sudo -iu enterprisedb psql bdrdb
```

Create the third node

Call the `bdr.create_node` function to create this node, passing it the node name and a connection string that other nodes can use to connect to it.

```
select bdr.create_node('node-three','host=host-three dbname=bdrdb port=5444');
```

Join the third node to the cluster

Using `bdr.join_node_group`, you can ask node-three to join node-one's `dc1` group. The function takes as a first parameter the connection string of a node already in the group and the group name as a second parameter.

```
select bdr.join_node_group('host=host-one dbname=bdrdb port=5444','dc1');
```

A PGD cluster is now created.

8.1.1.6 Step 6 - Checking the cluster

Checking the cluster

With the cluster up and running, it's worthwhile to run some basic checks to see how effectively it's replicating.

The following example shows one quick way to do this, but you must ensure that any testing you perform is appropriate for your use case.

- **Preparation**
  - Ensure the cluster is ready:
    - Log in to the database on host-one/node-one.
    - Run `select bdr.wait_slot_confirm_lsn(NULL, NULL);`.
    - When the query returns, the cluster is ready.
- **Create data** The simplest way to test that the cluster is replicating is to log in to one node, create a table, and populate it.
  - On node-one, create a table:

    ```
    CREATE TABLE quicktest ( id SERIAL PRIMARY KEY, value INT );
    ```

  - On node-one, populate the table:

    ```
    INSERT INTO quicktest (value) SELECT random()*10000 FROM
    generate_series(1,10000);
    ```

  - On node-one, monitor performance:

    ```
    select * from bdr.node_replication_rates;
    ```

  - On node-one, get a sum of the value column (for checking):

    ```
    select COUNT(*),SUM(value) from quicktest;
    ```

- **Check data**
  - Log in to node-two. Log in to the database on host-two/node-two.
  - On node-two, get a sum of the value column (for checking):

    ```
    select COUNT(*),SUM(value) from quicktest;
    ```

  - Compare with the result from node-one.
  - Log in to node-three. Log in to the database on host-three/node-three.
  - On node-three, get a sum of the value column (for checking):

    ```
    select COUNT(*),SUM(value) from quicktest;
    ```

  - Compare with the result from node-one and node-two.

Worked example

Preparation

Log in to host-one's Postgres server.

```
ssh admin@host-one
sudo -iu enterprisedb psql bdrdb
```

This is your connection to PGD's node-one.

Ensure the cluster is ready

To ensure that the cluster is ready to go, run:

```
select bdr.wait_slot_confirm_lsn(NULL, NULL)
```

This query blocks while the cluster is busy initializing and returns when the cluster is ready.

In another window, log in to host-two's Postgres server:

```
ssh admin@host-two
sudo -iu enterprisedb psql bdrdb
```

Create data

On node-one, create a table

Run:

```
CREATE TABLE quicktest ( id SERIAL PRIMARY KEY, value INT );
```

On node-one, populate the table

```
INSERT INTO quicktest (value) SELECT random()*10000 FROM generate_series(1,10000);
```

This command generates a table of 10000 rows of random values.

On node-one, monitor performance

As soon as possible, run:

```
select * from bdr.node_replication_rates;
```

The command shows statistics about how quickly that data was replicated to the other two nodes:

```
bdrdb=# select * from bdr.node_replication_rates;
 peer_node_id | target_name | sent_lsn  | replay_lsn | replay_lag | replay_lag_bytes | replay_lag_size |
apply_rate | catchup_interv
al
--------------+-------------+-----------+------------+------------+------------------+-----------------+-----------
--------------
---
   1954860017 | node-three  | 0/DDAA908 | 0/DDAA908  | 00:00:00   |                0 | 0 bytes         |      13682
| 00:00:00
   2299992455 | node-two    | 0/DDAA908 | 0/DDAA908  | 00:00:00   |                0 | 0 bytes         |      13763
| 00:00:00
(2 rows)
```

And it's already replicated.

On node-one get a checksum

Run:

```
select COUNT(*),SUM(value) from quicktest;
```

This command gets some values from the generated data:

```
bdrdb=# select COUNT(*),SUM(value) from quicktest;
```

output

```
 count  |    sum
--------+-----------
 100000 | 498884606
(1 row)
```

Check data

Log in to host-two's Postgres server

```
ssh admin@host-two
sudo -iu enterprisedb psql bdrdb
```

This is your connection to PGD's node-two.

On node-two, get a checksum

Run:

```
select COUNT(*),SUM(value) from quicktest;
```

This command gets node-two's values for the generated data:

```
bdrdb=# select COUNT(*),SUM(value) from quicktest;
```

output

```
 count  |    sum
--------+-----------
 100000 | 498884606
(1 row)
```

Compare with the result from node-one

The values are identical.

You can repeat the process with node-three or generate new data on any node and see it replicate to the other nodes.

Log in to host-three's Postgres server

```
ssh admin@host-two
sudo -iu enterprisedb psql bdrdb
```

This is your connection to PGD's node-three.

On node-three, get a checksum

Run:

```
select COUNT(*),SUM(value) from quicktest;
```

This command gets node-three's values for the generated data:

```
bdrdb=# select COUNT(*),SUM(value) from quicktest;
```

output

```
 count  |    sum
--------+-----------
 100000 | 498884606
(1 row)
```

Compare with the result from node-one and node-two

The values are identical.

8.1.1.7 Step 7 - Configure proxies

Configure proxies

PGD can use proxies to direct traffic to one of the cluster's nodes, selected automatically by the cluster. There are performance and availability reasons for using a proxy:

- Performance: By directing all traffic (in particular, write traffic) to one node, the node can resolve write conflicts locally and more efficiently.
- Availability: When a node is taken down for maintenance or goes offline for other reasons, the proxy can direct new traffic to a new write leader that it selects.

It's best practice to configure PGD Proxy for clusters to enable this behavior.

Configure the cluster for proxies

To set up a proxy, you need to first prepare the cluster and subgroup the proxies will be working with by:

- Logging in and setting the `enable_raft` and `enable_proxy_routing` node group options to `true` for the subgroup. Use `bdr.alter_node_group_option`, passing the subgroup name, option name, and new value as parameters.
- Create as many uniquely named proxies as you plan to deploy using `bdr.create_proxy` and passing the new proxy name and the subgroup to attach it to. The `bdr.create_proxy` does not create a proxy, but creates a space for a proxy to register itself with the cluster. The space contains configuration values which can be modified later. Initially it is configured with default proxy options such as setting the `listen_address` to `0.0.0.0`.
- Configure proxy routes to each node by setting route_dsn for each node in the subgroup. The route_dsn is the connection string that the proxy should use to connect to that node. Use `bdr.alter_node_option` to set the route_dsn for each node in the subgroup.
- Create a pgdproxy user on the cluster with a password or other authentication.

Configure each host as a proxy

Once the cluster is ready, you need to configure each host to run pgd-proxy:

- Create a pgdproxy local user.
- Create a `.pgpass` file for that user that allows the user to log into the cluster as pgdproxy.
- Modify the systemd service file for pgdproxy to use the pgdproxy user.
- Create a proxy config file for the host that lists the connection strings for all the nodes in the subgroup, specifies the name for the proxy to use when fetching proxy options like `listen_address` and `listen_port`.
- Install that file as `/etc/edb/pgd-proxy/pgd-proxy-config.yml`.
- Restart the systemd service and check its status.
- Log in to the proxy and verify its operation.

Further detail on all these steps is included in the worked example.

Worked example

Preparing for proxies

For proxies to function, the `dc1` subgroup must enable Raft and routing.

Log in to any node in the cluster, using psql to connect to the `bdrdb` database as the enterprisedb user. Execute:

```
SELECT bdr.alter_node_group_option('dc1', 'enable_raft', 'true');
SELECT bdr.alter_node_group_option('dc1', 'enable_proxy_routing', 'true');
```

You can use the `bdr.node_group_summary` view to check the status of options previously set with `bdr.alter_node_group_option()`:

```
SELECT node_group_name, enable_proxy_routing,
enable_raft
            FROM bdr.node_group_summary
           WHERE parent_group_name IS NOT NULL;
```

```
                                                output
 node_group_name | enable_proxy_routing | enable_raft
-----------------+----------------------+-------------
 dc1             | t                    | t
(1 row)

bdrdb=#
```

Next, create a PGD proxy within the cluster using the `bdr.create_proxy` function. This function takes two parameters: the proxy's unique name and the group you want it to be a proxy for.

In this example, you want a proxy on each host in the `dc1` subgroup:

```
SELECT bdr.create_proxy('pgd-proxy-one','dc1');
SELECT bdr.create_proxy('pgd-proxy-two','dc1');
SELECT bdr.create_proxy('pgd-proxy-three','dc1');
```

You can use the `bdr.proxy_config_summary` view to check that the proxies were created:

```
SELECT proxy_name,
node_group_name
             FROM bdr.proxy_config_summary;
```

```
                                                output
   proxy_name    | node_group_name
-----------------+-----------------
 pgd-proxy-one   | dc1
 pgd-proxy-two   | dc1
 pgd-proxy-three | dc1

 bdrdb=#
```

Create a pgdproxy user on the database

Create a user named pgdproxy and give it a password. This example uses `proxysecret`.

On any node, log into the `bdrdb` database as enterprisedb/postgres.

```
CREATE USER pgdproxy PASSWORD 'proxysecret';
GRANT bdr_superuser TO pgdproxy;
```

Configure proxy routes to each node

Once a proxy has connected, it gets its dsn values (connection strings) from the cluster. The cluster needs to know the connection details that a proxy should use for each node in the subgroup. This is done by setting the `route_dsn` option for each node to a connection string that the proxy can use to connect to that node.

Please note that when a proxy starts, it gets the initial dsn from the proxy's config file. The route_dsn value set in this step and in config file should match.

On any node, log into the bdrdb database as enterprisedb/postgres.

```
SELECT bdr.alter_node_option('node-one', 'route_dsn', 'host=host-one dbname=bdrdb port=5444
user=pgdproxy');
SELECT bdr.alter_node_option('node-two', 'route_dsn', 'host=host-two dbname=bdrdb port=5444
user=pgdproxy');
SELECT bdr.alter_node_option('node-three', 'route_dsn', 'host=host-three dbname=bdrdb port=5444
user=pgdproxy');
```

Note that the endpoints in this example specify `port=5444`. This is necessary for EDB Postgres Advanced Server instances. For EDB Postgres Extended and community PostgreSQL, you can omit this.

Create a pgdproxy user on each host

```
sudo adduser pgdproxy
```

This user needs credentials to connect to the server. Create a `.pgpass` file with the `proxysecret` password in it. Then lock down the `.pgpass` file so it's accessible only by its owner.

```
echo -e "*:*:*:pgdproxy:proxysecret" | sudo tee /home/pgdproxy/.pgpass
sudo chown pgdproxy /home/pgdproxy/.pgpass
sudo chmod 0600 /home/pgdproxy/.pgpass
```

Configure the systemd service on each host

Switch the service file from using root to using the pgdproxy user.

```
sudo sed -i s/root/pgdproxy/ /usr/lib/systemd/system/pgd-proxy.service
```

Reload the systemd daemon.

```
sudo systemctl daemon-reload
```

Create a proxy config file for each host

The proxy configuration file is slightly different for each host. It's a YAML file that contains a cluster object. The cluster object has three properties:

- The name of the PGD cluster's top-level group (as `name`)
- An array of endpoints of databases (as `endpoints`)
- The proxy definition object with a name and endpoint (as `proxy`)

The first two properties are the same for all hosts:

```
cluster:
  name: pgd
  endpoints:
    - "host=host-one dbname=bdrdb port=5444 user=pgdproxy"
    - "host=host-two dbname=bdrdb port=5444 user=pgdproxy"
    - "host=host-three dbname=bdrdb port=5444 user=pgdproxy"
```

Remember that host-one, host-two, and host-three are the systems on which the cluster nodes (node-one, node-two, node-three) are running. You use the name of the host, not the node, for the endpoint connection.

Again, note that the endpoints in this example specify `port=5444`. This is necessary for EDB Postgres Advanced Server instances. For EDB Postgres Extended and community PostgreSQL, you can set this to `port=5432`.

The third property, `proxy`, has a `name` property. The `name` property is a name created with `bdr.create_proxy` earlier, and it's different on each host. A proxy can't be on the same port as the Postgres server and, ideally, should be on a commonly used port different from direct connections, even when no Postgres server is running on the host. Typically, you use port 6432 for PGD proxies.

```
  proxy:
    name: pgd-proxy-one
```

In this case, using `localhost` in the endpoint specifies that this proxy will listen on the host where the proxy is running.

Install a PGD proxy configuration on each host

For each host, create the `/etc/edb/pgd-proxy` directory:

```
sudo mkdir -p /etc/edb/pgd-proxy
```

Then, on each host, write the appropriate configuration to the `pgd-proxy-config.yml` file in the `/etc/edb/pgd-proxy` directory.

For this example, you can run this on host-one to create the file:

```
cat <<EOF | sudo tee /etc/edb/pgd-proxy/pgd-proxy-config.yml
cluster:
  name: pgd
  endpoints:
    - "host=host-one dbname=bdrdb port=5444 user=pgdproxy"
    - "host=host-two dbname=bdrdb port=5444 user=pgdproxy"
    - "host=host-three dbname=bdrdb port=5444 user=pgdproxy"
  proxy:
    name: pgd-proxy-one
EOF
```

Restart the service

On each host where the proxy is being installed, restart the pgd-proxy service.

```
sudo systemctl restart pgd-proxy
```

Confirm it's running correctly:

```
sudo systemctl status pgd-proxy
```

When running, it shows `Active: (running)` in the opening details.

Test the proxy

At this point, connecting to the PGD Proxy port on any host in the cluster results in the connection being routed to the current write lead node.

For example, and assuming you've installed the proxy on all three hosts, then connecting to the proxy on host-three results in the connection being routed to node-one.

This example passes connection details, using the `-d` flag of psql with the hostname for the proxy you just configured and the proxy port number:

```
sudo -iu enterprisedb psql -d "host=host-three dbname=bdrdb port=6432"
```

output

```
psql (16.1.0, server 16.1.0)
Type "help" for help.

bdrdb=#
```

Once connected to the proxy, you can query the server to find out which node the proxy connected you to:

```
SELECT node_name FROM bdr.local_node_summary;
```

output

```
 node_name
-----------
 node-one
(1 row)
bdrdb#
```

You should have connected to the current write leader of the subgroup. You can confirm that by querying which node is the write leader for the subgroup you're connected to:

```
SELECT node_group_name, write_lead FROM
bdr.node_group_routing_summary;
```

output

```
 node_group_name | write_lead
-----------------+------------
 dc1             | node-one
(1 row)

bdrdb=#
```

And the write_lead is node-one, too, so you confirm you're being proxy-connected to the write leader.

8.1.1.8 Step 8 - Using PGD CLI

Using PGD CLI

The PGD CLI command uses a configuration file to work out the hosts to connect to. There are options that allow you to override this to use alternative configuration files or explicitly point at a server. But, by default, PGD CLI looks for a configuration file in preset locations.

The connection to the database is authenticated in the same way as other command line utilities, like the psql command, are authenticated.

Unlike other commands, PGD CLI doesn't interactively prompt for your password. Therefore, you must pass your password using one of the following methods:

- Adding an entry to your `.pgpass` password file, which includes the host, port, database name, user name, and password
- Setting the password in the `PGPASSWORD` environment variable
- Including the password in the connection string

We recommend the first option, as the other options don't scale well with multiple database clusters, or they compromise password confidentiality.

Configuring and connecting PGD CLI

- Ensure PGD CLI is installed.
  - If PGD CLI was already installed, move to the next step.
  - For any system, repeat the configure repositories step on that system.
  - Then run the package installation command appropriate for that platform.
    - RHEL and derivatives: `sudo dnf install edb-pgd5-cli`
    - Debian, Ubuntu, and derivatives: `sudo apt-get install edb-pgd5-cli`

- Create a configuration file.
  - This is a YAML file that specifies the cluster and endpoints for PGD CLI to use.

- Install the configuration file.
  - Copy the YAML configuration file to a default config directory `/etc/edb/pgd-cli/` as `pgd-cli-config.yml`.
  - Repeat this process on any system where you want to run PGD CLI.

- Run pgd-cli.

Use PGD CLI to explore the cluster

- Check the health of the cluster with the `check-health` command.
- Show the nodes in the cluster with the `show-nodes` command.
- Show the proxies in the cluster with the `show-proxies` command.
- Show the groups in the cluster with the `show-groups` command.
- Set a group option with the `set-group-options` command.
- Switch write leader with the `switchover` command.

For more details about these commands, see the worked example that follows.

Also consult the PGD CLI documentation for details of other configuration options and a full command reference.

Worked example

Ensure PGD CLI is installed

In this worked example, you configure and use PGD CLI on host-one, where you've already installed Postgres and PGD. You don't need to install PGD CLI again.

Create a configuration file

The PGD CLI configuration file is similar to the PGD Proxy configuration file. It's a YAML file that contains a cluster object. This has two properties:

- The name of the PGD cluster's top-level group (as `name`)
- An array of endpoints of databases (as `endpoints`)

```
cluster:
  name: pgd
  endpoints:
    - host=host-one dbname=bdrdb port=5444
    - host=host-two dbname=bdrdb port=5444
    - host=host-three dbname=bdrdb port=5444
```

Note that the endpoints in this example specify `port=5444`. This is necessary for EDB Postgres Advanced Server instances. For EDB Postgres Extended and community PostgreSQL, you can omit this.

Install the configuration file

Create the PGD CLI configuration directory:

```
sudo mkdir -p /etc/edb/pgd-cli
```

Then, write the configuration to the `pgd-cli-config.yml` file in the `/etc/edb/pgd-cli` directory.

For this example, you can run this on host-one to create the file:

```
cat <<EOF | sudo tee /etc/edb/pgd-cli/pgd-cli-config.yml
cluster:
  name: pgd
  endpoints:
    - host=host-one dbname=bdrdb port=5444
    - host=host-two dbname=bdrdb port=5444
    - host=host-three dbname=bdrdb port=5444
EOF
```

You can repeat this process on any system where you need to use PGD CLI.

Run PGD CLI

With the configuration file in place, and logged in as the enterprisedb system user, you can run pgd-cli. For example, you can use the `show-nodes` command to list the nodes in your cluster and their status:

```
pgd show-nodes
```

```
output
 Node       Node ID    Group Type Current State Target State Status Seq ID
 ----       -------    ----- ---- ------------- ------------ ------ ------
 node-one   2824718320 dc1   data ACTIVE        ACTIVE       Up     1
 node-three 1954860017 dc1   data ACTIVE        ACTIVE       Up     3
 node-two   2299992455 dc1   data ACTIVE        ACTIVE       Up     2
```

Using PGD CLI to explore the cluster

Once PGD CLI is configured, you can use it to get PGD-level views of the cluster.

Check the health of the cluster

The `check-health` command provides a quick way to view the health of the cluster:

```
pgd check-health
```

```
output
 Check      Status Message
 -----      ------ -------
 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
```

Show the nodes in the cluster

As previously seen, the `show-nodes` command lists the nodes in the cluster:

```
pgd show-nodes
```

```
output
 Node       Node ID    Group Type Current State Target State Status Seq ID
 ----       -------    ----- ---- ------------- ------------ ------ ------
 node-one   2824718320 dc1   data ACTIVE        ACTIVE       Up     1
 node-three 1954860017 dc1   data ACTIVE        ACTIVE       Up     3
 node-two   2299992455 dc1   data ACTIVE        ACTIVE       Up     2
```

This view shows the group the node is a member of and its current status. To find out what versions of PGD and Postgres are running on the nodes, use `show-version`:

```
pgd show-version
```

```
output
 Node       BDR Version Postgres Version
 ----       ----------- ----------------
 node-one   5.3.0       16.1.0
 node-three 5.3.0       16.1.0
 node-two   5.3.0       16.1.0
```

Show the proxies in the cluster

You can view the configured proxies, with their groups and ports, using `show-proxies`:

```
pgd show-proxies
```

```
                                                output
 Proxy           Group Listen Addresses Listen Port
-----           ----- ---------------- -----------
pgd-proxy-one   dc1   [0.0.0.0]        6432
pgd-proxy-three dc1   [0.0.0.0]        6432
pgd-proxy-two   dc1   [0.0.0.0]        6432
```

Show the groups in the cluster

Finally, the `show-groups` command for PGD CLI shows which groups are configured, and more:

```
pgd show-node_groups
```

```
                                                output
 Group Group ID   Type   Parent Group Location Raft Routing Write Leader
----- --------   ----   ------------ -------- ---- ------- ------------
pgd   1850374637 global                        true false
dc1   4269540889 data   pgd                   true true    node-one
```

This command shows:

- The groups
- Their types
- Their parent group
- The group's location
- Whether Raft consensus is enabled
- Whether the group is routing connections and, if it is, the node that's write leader for that

The location is descriptive metadata, and so far you haven't set it. You can use PGD CLI to do that.

Set a group option

You can set group options using PGD CLI, too, using the `set-group-options` command. This requires a `--group-name` flag to set the group for this change to affect and an `--option` flag with the setting to change. If you wanted to set the `dc1` group's location to `London`, you would run:

```
pgd set-group-options --group-name dc1 --option "location=London"
```

```
                                                output
 group options updated successfully
```

You can verify that with `show-groups`:

```
pgd show-node_groups
```

```
                                                output
 Group Group ID   Type   Parent Group Location Raft Routing Write Leader
----- --------   ----   ------------ -------- ---- ------- ------------
pgd   1850374637 global                        true false
dc1   4269540889 data   pgd          London    true true    node-one
```

Switching write leader

If you need to change write leader in a group, to enable maintenance on a host, PGD CLI offers the `switchover` command. It takes a `--group-name` flag with the group the node exists in and a `--node-name` flag with the name of the node to switch to. You can then run:

```
pgd switchover --group-name dc1 --node-name node-two
```

output

```
switchover is complete
```

And you can verify that with `show-groups`:

```
pgd show-groups
```

output

```
 Group Group ID   Type   Parent Group Location Raft Routing Write Leader
------ --------   ----   ------------ -------- ---- ------- ------------
pgd    1850374637 global                       true false
dc1    4269540889 data   pgd          London   true true    node-two
```

More details on the available commands in PGD CLI are available in the PGD CLI command reference.

8.2 Deployment and management with TPA

TPA (Trusted Postgres Architect) is a standard automated way of installing PGD and Postgres on physical and virtual machines, both self-hosted and in the cloud (with AWS EC2).

> **Get started with TPA and PGD quickly**
>
> If you want to experiment with a local deployment as quickly as possible, you can deploying an EDB Postgres Distributed example cluster on Docker to configure, provision, and deploy a PGD 5 Always-on cluster on Docker.
>
> If deploying to the cloud is your aim, you can deploying an EDB Postgres Distributed example cluster on AWS to get a PGD 5 cluster on your own Amazon account.
>
> If you want to run on your own Linux systems or VMs, you can use also use TPA to deploy EDB Postgres Distributed directly to your own Linux hosts

This section covers how to use TPA to deploy and administer EDB Postgres Distributed.

- Deploying with TPA works through the steps needed to:
  - Install TPA.
  - Use TPA to create a configuration.
  - Deploy the configuration with TPA.

The installing section provides an example cluster that will be used in future examples.

You can also perform a rolling major version upgrade with PGD administered by TPA.

8.2.1 Deploying PGD using TPA

The standard way of automatically deploying EDB Postgres Distributed in a self-managed setting is to use EDB's deployment tool: Trusted Postgres Architect (TPA). This applies to physical and virtual machines, both self-hosted and in the cloud (EC2),

> **Get started with TPA and PGD quickly**
>
> If you want to experiment with a local deployment as quickly as possible, you can deploy an EDB Postgres Distributed example cluster on Docker to configure, provision, and deploy a PGD 5 Always-on cluster on Docker.
>
> If deploying to the cloud is your aim, you can deploy an EDB Postgres Distributed example cluster on AWS to get a PGD 5 cluster on your own Amazon account.
>
> If you want to run on your own Linux systems or VMs, you can also use TPA to deploy EDB Postgres Distributed directly to your own Linux hosts.

Prerequisite: Install TPA

Before you can use TPA to deploy PGD, you must install TPA. Follow the installation instructions in the Trusted Postgres Architect documentation before continuing.

At the highest level, using TPA to deploy PGD involves the following steps:

1: Use TPA to create a configuration for your PGD cluster.

2: Provision, deploy, and test your PGD cluster.

8.2.1.1 Configuring a PGD cluster with TPA

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> [options]
```

The available configuration options include:

Flags	Description
`--architecture`	Required. Set to `PGD-Always-ON` for EDB Postgres Distributed deployments.
`--postgresql <version>` or `--edb-postgres-advanced <version>` or `--edb-postgres-extended <version>`	Required. Specifies the distribution and version of Postgres to use. For more details, seeCluster configuration: Postgres flavour and version.
`--redwood` or `--no-redwood`	Required when `--edb-postgres-advanced` flag is present. Specifies whether Oracle database compatibility features are desired.
`--location-names l1 l2 l3`	Required. Specifies the names of the locations to deploy PGD to.
`--data-nodes-per-location N`	Specifies the number of data nodes per location. Default is 3.
`--add-witness-node-per-location`	For an even number of data nodes per location, adds witness nodes to allow for local consensus. Enabled by default for 2-data-node locations.
`--add-proxy-nodes-per-location`	Specifies whether to separate PGD proxies from data nodes and how many to configure. By default one proxy is configured and cohosted for each data node.
`--pgd-proxy-routing global|local`	Specifies whether PGD Proxy routing is handled on a global or local (per-location) basis.
`--add-witness-only-location loc`	Designates one of the cluster locations as witness-only (no data nodes are present in that location).
`--enable-camo`	Sets up a CAMO pair in each location. Works only with 2 data nodes per location.

More configuration options are listed in the TPA documentation for PGD-Always-ON.

For example:

```
[tpa]$ tpaexec configure ~/clusters/speedy \
        --architecture PGD-Always-ON \
        --platform aws \
        --edb-postgres-advanced 16 \
        --redwood \
        --location-names eu-west-1 eu-north-1 eu-central-1 \
        --data-nodes-per-location 3 \
        --pgd-proxy-routing global
```

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 options.

The command creates a directory named `~/clusters/speedy` and generates a configuration file named `config.yml` that follows the layout of the PGD-Always-ON architecture. You can use the `tpaexec configure --architecture PGD-Always-ON --help` command to see the values that are supported for the configuration options in this architecture.

In the example, the options select:

- An AWS deployment (`--platform aws`)
- EDB Postgres Advanced Server, version 16 and Oracle compatibility (`--edb-postgres-advanced 16` and `--redwood`)
- Three locations (`--location-names eu-west-1 eu-north-1 eu-central-1`)
- Three data nodes at each location (`--data-nodes-per-location 3`)
- Proxy routing policy of global (`--pgd-proxy-routing global`)

Common configuration options

Other configuration options include the following.

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 doesn't identify a person (for example, `postgres` , `ec2-user`), you must specify `--owner SomeId` to set an identifiable owner.

You can use your initials, "Firstname Lastname", or any text that identifies you uniquely.

Platform options

The default value for `--platform` is `aws` , which is the platform supported by the PGD-Always-ON architecture.

Specify `--region` to specify any existing AWS region that you have access to and that allows you to create the required number of instances. 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. However, depending on the architecture, there can be one or more subnets, and each subnet can be anywhere between a /24 and a /29.

Specify `--subnet` to use a particular subnet, for example, `--subnet 192.0.2.128/27` .

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 might 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 use on the cluster's instances. The value is case sensitive.

The selected platform determines the distributions that are available and the one that's 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`.

Repositories

When using TPA to deploy PDG 5 and later, TPA selects repositories from EDB Repos 2.0. All software is sourced from these repositories.

To use EDB Repos 2.0, you must use `export EDB_SUBSCRIPTION_TOKEN=xxx` before you run tpaexec. You can get your subscription token from the web interface.

Optionally, use `--edb-repositories repository …` to specify EDB repositories in addition to the default repository to install on each instance.

Software versions

By default, TPA uses the latest major version of Postgres. Specify `--postgres-version` to install an earlier supported major version, or specify both version and distribution using one of the flags described under Configure.

By default, TPA installs the latest version of every package, which is usually the desired behavior. However, in some testing scenarios, you might need 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 more packages. The former lists packages to install along with system packages. 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 isn't 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's not an error if the named packages can't be installed.

Hostnames

By default, `tpaexec configure` randomly selects as many hostnames as it needs from a preapproved list of several dozen names, which is 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 supplied 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 enough of them.

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.

8.2.1.2 Provisioning, deploying, and testing

Provision

> **Note**
>
> TPA now runs the `provision` command as part of the `deploy` command. The `provision` command is still available for use, but you don't need to run it separately.

The `tpaexec provision` command creates instances and other resources required by the cluster. The details of the process depend on the architecture (for example, PGD-Always-ON) and platform (for example, AWS) that you selected while configuring the cluster.

For example, given AWS access with the necessary privileges, TPA 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, you can use them as if they were created by TPA.

You aren't restricted to a single platform. You can spread your cluster out across some AWS instances in multiple regions and some on-premises 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 TPA can access using SSH (with sudo to root).

Deploy

The `tpaexec deploy` command installs and configures Postgres and other software on the provisioned servers. This includes setting up replication, backups, and so on. TPA can create the servers, but it doesn't matter who created them so long as SSH and sudo access are available.

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's working as expected.

At the end of the testing stage, you have a fully functioning cluster.

For more information, see Trusted Postgres Architect.

8.3 Deploying and configuring PGD on Kubernetes

EDB Postgres Distributed for Kubernetes is a Kubernetes operator designed, developed, and supported by EDB. It covers the full lifecycle of highly available Postgres database clusters with a multi-master architecture, using PGD replication. It's based on the open source CloudNativePG operator and provides additional value, such as compatibility with Oracle using EDB Postgres Advanced Server, Transparent Data Encryption (TDE) using EDB Postgres Extended or Advanced Server, and additional supported platforms including IBM Power and OpenShift.

This section covers how to deploy and configure EDB Postgres Distributed using the Kubernetes operator.

- Quick start in the PGD for Kubernetes documentation works through the steps needed to:
    - Create a Kind/Minikube cluster.
    - Install Helm and the Helm chart for PGD for Kubernetes.
    - Create a simple configuration file for a PGD cluster.
    - Deploy a PGD cluster from that simple configuration file.
- Installation and upgrade provides detailed instructions for installing and upgrading PGD for Kubernetes.

8.4 Deploying and configuring PGD on EDB BigAnimal

EDB BigAnimal is a fully managed database-as-a-service with built-in Oracle compatibility. It runs in your cloud account or BigAnimal's cloud account, where it's operated by our Postgres experts. EDB BigAnimal makes it easy to set up, manage, and scale your databases. The addition of distributed high-availability support powered by EDB Postgres Distributed (PGD) enables single- and multi-region Always-on clusters.

This section covers how to work with EDB Postgres Distributed when deployed on BigAnimal.

- Creating a distributed high-availability cluster in the BigAnimal documentation works through the steps needed to:
    - Prepare your cloud environment for a distributed high-availability cluster.
    - Sign in to BigAnimal.
    - Create a distributed high-availability cluster, including:
        - Creating and configuring a data group.
        - Optionally creating and configuring a second data group in a different region.

9 Application use

Developing an application with PGD is mostly the same as working with any PostgreSQL database. What's different, though, is that you need to be aware of how your application interacts with replication. You need to know how PGD behaves with applications, the SQL that is and isn't replicated, how different nodes are handled, and other important information.

- Application behavior looks at how PGD replication appears to an application, such as:
    - The commands that are replicated
    - The commands that run locally
    - When row-level locks are acquired
    - How and where triggers fire
    - Large objects
    - Toast
- DML and DDL replication shows the differences between the two classes of SQL statements and how PGD handles replicating them. It also looks at the commands PGD doesn't replicate at all.
- Nodes with differences examines how PGD works with configurations where there are differing table structures and schemas on replicated nodes. Also covered is how to compare between such nodes with LiveCompare and how differences in PostgreSQL versions running on nodes can be handled.
- Application rules offers some general rules for applications to avoid data anomalies.
- Timing considerations shows how the asynchronous/synchronous replication might affect an application's view of data and notes functions to mitigate stale reads.
- Extension usage explains how to select, install, and configure extensions on PGD.
- Table access methods (TAMs) notes the TAMs available with PGD and how to enable them.
- Feature compatibility shows which server features work with which commit scopes and which commit scopes can be daisy chained together.

9.1 Application behavior

Much of PGD's replication behavior is transparent to applications. Understanding how it achieves that and the elements that aren't transparent is important to successfully developing an application that works well with PGD.

Replication behavior

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

PGD, by default, replicates 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.

You can replicate INSERT without any preconditions.

For updates and deletes to replicate on other nodes, PGD must be able to identify the unique rows affected. PGD 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's live, valid, not deferred, and doesn't have expressions or WHERE clauses. Otherwise, a sequential scan is used.

Truncate

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 details and examples. This works correctly if all affected tables are part of the same subscription. But if some tables to truncate 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

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. You must either handle these or avoid them. Various avoidance mechanisms are possible, discussed in Conflicts.

Sequences

Sequences need special handling, described in Sequences. This is because in a cluster, sequences must be global to avoid nodes creating conflicting values. Global sequences are available with global locking to ensure integrity.

Binary objects

Binary data in BYTEA columns is replicated normally, allowing "blobs" of data up to 1 GB. Use of the PostgreSQL "large object" facility isn't supported in PGD.

Rules

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

Base tables only

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.

Partitioned tables

PGD supports partitioned tables transparently, meaning that you can add a partitioned table to a replication set and changes that involve any of the partitions are replicated downstream.

Triggers

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 [, ...]`)

PGD 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, use any of these techniques:

- Resolve object references clearly using only the default search_path.
- Always use fully qualified references to objects, for example, `schema.objectname`.
- Set the search path for a function using `ALTER FUNCTION ... SET search_path = ...` for the functions affected.

PGD assumes that there are no issues related to text or other collatable datatypes, that is, all collations in use are available on all nodes, and the default collation is the same on all nodes. Replicating changes uses equality searches to locate Replica Identity values, so this doesn't have any effect except where unique indexes are explicitly defined with nonmatching collation qualifiers. Row filters might be affected by differences in collations if collatable expressions were used.

Toast

PGD 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.

Other restrictions

PGD can't work correctly if Replica Identity columns are marked as external.

PostgreSQL allows CHECK() constraints that contain volatile functions. Since PGD reexecutes CHECK() constraints on apply, any subsequent reexecution that doesn't return the same result as before causes data divergence.

PGD doesn't restrict the use of foreign keys. Cascading FKs are allowed.

9.2 DML and DDL replication and nonreplication

The two major classes of SQL statement are DML and DDL.

- DML is the data modification language and is concerned with the SQL statements that modify the data stored in tables. It includes UPDATE, DELETE, and INSERT.
- DDL is the data definition language and is concerned with the SQL statements that modify how the data is stored. It includes CREATE, ALTER, and DROP.

PGD handles each class differently.

DML replication

PGD 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

DDL replication works differently from DML. For DDL, PGD 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. For details, see DDL replication.

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

Nonreplicated statements

Outside of those two classes are SQL commands that PGD, by design, doesn't replicate. None of the following user commands are replicated by PGD, 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 PGD doesn't yet improve on this.

CAMO and Eager Replication options don't allow the `NOTIFY` SQL command or the `pg_notify()` function.

9.3 Nodes with differences

Replicating between nodes with differences

By default, DDL is sent to all nodes. You can control this behavior, as described in DDL replication, and you can use it to create differences between database schemas across nodes. PGD 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's possible to replicate between tables with dissimilar partitioning definitions, such as a source that's 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.

PGD replicates data columns based on the column name. If a column has the same name but a different data type, PGD attempts to cast from the source type to the target type, if casts were defined that allow that.

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

If the target has missing columns from the source, then PGD 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 PGD 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. If not, it throws an error and halts replication.

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 doesn'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. To specify how to create an index on only a subset of nodes or just locally, see DDL replication .

Storage parameters, such as `fillfactor` and `toast_tuple_target`, can differ between nodes for a table without problems. An exception to that behavior is that 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 details.

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 on only a subset of nodes or locally.

Comparison between nodes with differences

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

Starting with LiveCompare 1.3, you can configure with `all_bdr_nodes` set. This setting saves you from clarifying all the relevant DSNs for each separate node in the cluster. An 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 and later 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 mechanism 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 any differences found.

Replicating between different release levels

The other difference between nodes that you might encounter is where there are different major versions of PostgreSQL on the nodes. PGD is designed to replicate between different major release versions. This feature is designed to allow major version upgrades without downtime.

PGD is also designed to replicate between nodes that have different versions of PGD 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. Doing so returns an error.

Both of these features might be affected by specific restrictions. See Release notes for any known incompatibilities.

9.4 General rules for applications

Background

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

Similarly, since PGD 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.

Rules for applications

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. However, use caution: while anomalies are unlikely, they aren't impossible. For example, you can reuse a row value 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.

9.5 Timing considerations and synchronous replication

Being asynchronous by default, peer nodes might lag behind. This behavior makes it possible for a client connected to multiple PGD 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 PGD as well. In addition, PGD provides multiple variants for more synchronous replication. See [Commit scopes(../commit-scopes) for an overview and comparison of all variants available and their different modes.

9.6 Using extensions with PGD

PGD and other PostgreSQL extensions

PGD is implemented as a PostgreSQL extension (see Supported Postgres database servers). It takes advantage of PostgreSQL's expandability and flexibility to modify low-level system behavior to provide multi-master replication.

In principle, extensions provided by community PostgreSQL, EDB Postgres Advanced Server, and third-party extensions can be used with PGD. However, the distributed nature of PGD means that you need to carefully consider and plan the extensions you select and install.

Extensions providing logical decoding

Extensions providing logical decoding, such as wal2json, may in theory work with PGD. However, there's no support for failover, meaning any WAL stream being read from such an extension can be interrupted.

Extensions providing replication or HA functionality

Any extension extending PostgreSQL with functionality related to replication or HA/failover is unlikely to work well with PGD and may even be detrimental to the health of the PGD cluster. We recommend avoiding these.

Supported extensions

These extensions are explicitly supported by PGD.

EDB Advanced Storage table access methods

The EDB Advanced Storage Pack provides a selection of table access methods (TAMs) implemented as extensions. The following TAMs are certified for use with PGD:

- Autocluster
- Refdata

For more details, see Table access methods.

pgaudit

PGD was modified to ensure compatibility with the pgaudit extension. See Postgres settings for configuration information.

Installing extensions

PostgreSQL extensions provide SQL objects, such as functions, datatypes, and, optionally, one or more shared libraries. These must be loaded into the PostgreSQL backend before you can install and use the extension.

> **Warning**
>
> The relevant extension packages must be available on all nodes in the cluster. Otherwise extension installation can fail and impact cluster stability.

If PGD is deployed using Trusted Postgres Architect, configure extensions using that tool. For details, see Adding Postgres extensions.

The following is relevant for manually configured PGD installations.

Configuring shared_preload_libraries

If an extension provides a shared library, include this library in the `shared_preload_libraries` configuration parameter before installing the extension.

`shared_preload_libraries` consists of a comma-separated list of extension names. It must include `bdr`. The order in which you specify other extensions generally doesn't matter. However if you're using the pgaudit extension, `pgaudit` must appear in the list before `bdr`.

Configure `shared_preload_libraries` on all nodes in the cluster before installing the extension with `CREATE EXTENSION`. You must restart PostgreSQL to activate the new configuration.

See also Postgres settings.

Installing the extension

Install the extension using the `CREATE EXTENSION` command. You need to do this on only one node in the cluster. PGD's DDL replication will ensure that it propagates to all other nodes.

> **Warning**
>
> Do not attempt to install extensions manually on each node by, for example, disabling DDL replication before executing `CREATE EXTENSION`.
>
> Do not use a command such as `bdr.replicate_ddl_command()` to execute `CREATE EXTENSION`.

9.7 Use of table access methods (TAMs) in PGD

The EDB Advanced Storage Pack provides a selection of table access methods (TAMs), available from EDB Postgres 15.0.

The following TAMs were certified for use with PGD 5.0:

- Autocluster
- Refdata

Usage of any other TAM is restricted until certified by EDB.

To use one of these TAMs on a PGD cluster, the appropriate extension library (`autocluster` and/or `refdata`) must be added to the `shared_preload_libraries` parameter on each node, and the PostgreSQL server restarted.

Once the extension library is present in `shared_preload_libraries` on all nodes in the cluster, the extension itself can be created with `CREATE EXTENSION autocluster;` or `CREATE EXTENSION refdata;` . The `CREATE EXTENSION` command only needs to be executed on one node; it will be replicated to the other nodes in the cluster.

After you create the extension, use `CREATE TABLE test USING autocluster;` or `CREATE TABLE test USING refdata;` to create a table with the specified TAM. These commands replicate to all PGD nodes in the cluster.

For more information on these table access methods, see:

- Autocluster example
- Refdata example

9.8 Feature compatibility

Server feature/commit scope interoperability

Not all server features work with all commit scopes. This table shows the ones that interoperate.

	Async (default)	Parallel Apply	Transaction Streaming	Single Decoding Worker
Group Commit				
CAMO				
Lag Control				
Synchronous Commit				

Legend: Not applicable | Does not interoperate | Interoperates

Notes

: The Async column in the table represents PGD without a synchronous commit scope in use. Lag Control isn't a synchronous commit scope. It's a controlling commit scope and is therefore available with asynchronous operations.

: Attempting to use Group Commit and Transaction Streaming presents a warning. The warning suggests that you disable transaction streaming, and the transaction appears to take place. In the background, Group Commit was disabled to allow the transaction to occur.

Commit scope/commit scope interoperability

Although you can't mix commit scopes, you can combine rules with an `AND` operator. This table shows where commit scopes can be combined.

	Group Commit	CAMO	Lag Control	Synchronous Commit
Group Commit				
CAMO				
Lag Control				
Synchronous Commit				

Legend: Not applicable | Does not combine | Combines

Notes

Each commit scope implicitly works with itself.

10 DDL replication

DDL stands for data definition language, the subset of the SQL language that creates, alters, and drops database objects.

PGD provides automatic DDL replication, which makes certain DDL changes easier. With automatic replication, you don't have to manually distribute the DDL change to all nodes and ensure that they're consistent.

This section looks at how DDL replication is handled in PGD.

- Overview provides a general outline of what PGD's DDL replication is capable of.
- Replication options looks at the options for controlling replication.
- Locking examines how DDL replication uses locks to safely replicate DDL.
- Managing DDL with PGD replication gives best practice advice on why and how to limit the impact of DDL changes so they don't overly affect the smooth running of the cluster.
- Command handling is a reference to all DDL commands, the locks they take, and any special handling involved in their use and execution.
- DDL role manipulation notes issues around manipulating roles over multiple databases in a cluster.
- Workarounds gives a range of options for handling situations where DDL replication may present restrictions, such as altering columns, constraints, and types.
- DDL-like PGD functions details the PGD functions that behave like DDL and therefore behave in a similar way and are subject to similar restrictions.

10.1 DDL overview

DDL stands for data definition language, the subset of the SQL language that creates, alters, and drops database objects.

Replicated DDL

For operational convenience and correctness, PGD 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're consistent.

In the default replication set, DDL is replicated to all nodes by default.

Differences from PostgreSQL

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

The main difference from table replication is that DDL replication doesn't replicate the result of the DDL. Instead, it replicates the statement. 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.

Executing DDL on PGD systems

A PGD 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 PGD. 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.

PGD offers three levels of protection against those problems:

`ddl_locking = 'all'` is the strictest option and is best when DDL might execute from any node concurrently and you want to ensure correctness. This is the default.

`ddl_locking = 'dml'` is an option that is safe only when you execute DDL from one node at any time. Use this setting only if you can completely control where DDL is executed. Executing DDL from a single node ensures that there are no inter-node conflicts. Intra-node conflicts are already handled by PostgreSQL.

`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 `postgres.conf` configuration 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.

10.2 DDL replication options

The `bdr.ddl_replication` parameter specifies replication behavior.

`bdr.ddl_replication = on` is the default. This setting replicates DDL to the default replication set, which by default means all nodes. Non-default 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 function can be helpful if you want to run DDL commands when a node is down. It's also helpful 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
               'all');              -- ddl_locking to apply
```

While we don't recommend it, you can skip automatic DDL replication and execute it manually on each node using the `bdr.ddl_replication` configuration parameter.

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

When set, it makes PGD 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 PGD group to stop replicating if conflicting DDL or DML executes concurrently.

Only the bdr_superuser or superuser can set the `bdr.ddl_replication` parameter. It can also be set in the `postgres.conf` configuration file.

10.3 DDL locking details

Two kinds of locks enforce correctness of replicated DDL with PGD: the global DDL lock and the global DML lock.

The global DDL lock

A global DDL lock is used only when `ddl_locking = 'all'`. This kind of lock prevents any other DDL from executing on the cluster while each DDL statement runs. This behavior ensures full correctness in the general case but is too strict for many simple cases. PGD 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 PGD node executing DDL contacts the other nodes in a PGD 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 agree, 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 catches 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.

A global DDL lock must be granted by a majority of data and witness nodes, where a majority is N/2+1 of the eligible nodes. Subscriber-only nodes aren't eligible to participate.

The global DML lock

Known as a global DML lock or relation DML lock, this kind of lock is used when either `ddl_locking = all` 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. These locks ensure that no DDL executes while changes are in the queue that might cause replication to halt with an error.

To acquire the global DML lock on a table, the PGD node executing the DDL contacts all other nodes in a PGD 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, unlike with data changes, schema changes affecting row representation and constraints 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.

All eligible data nodes must agree to grant a global DML lock before the lock is granted. Witness and subscriber-only nodes aren't eligible to participate.

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

Specify locking behavior with the `bdr.ddl_locking` parameter, as explained in Executing DDL on PGD systems:

- `ddl_locking = all` 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 PGD functions make DDL changes. For those functions, DDL locking behavior applies, which is noted in the documentation 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're promoted.

Some PGD 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 PGD 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.

10.4 Managing DDL with PGD replication

Minimizing the impact of DDL

Minimizing the impact of DDL is good operational advice for any database. These points become even more important with PGD:

- 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 alternative 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 3.6 and earlier but not with PGD 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 PGD 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 PGD 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.

PGD 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 PGD, 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 executing, raising a `feature_not_supported` (SQLSTATE `0A000`) error.

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

10.5 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 matrix

Command	Allowed	Replicated	Lock
ALTER AGGREGATE	Y	Y	DDL
ALTER CAST	Y	Y	DDL
ALTER COLLATION	Y	Y	DDL
ALTER CONVERSION	Y	Y	DDL
ALTER DATABASE	Y	N	N
ALTER DATABASE LINK	Y	Y	DDL
ALTER DEFAULT PRIVILEGES	Y	Y	DDL
ALTER DIRECTORY	Y	Y	DDL
ALTER DOMAIN	Y	Y	DDL
ALTER EVENT TRIGGER	Y	Y	DDL
ALTER EXTENSION	Y	Y	DDL
ALTER FOREIGN DATA WRAPPER	Y	Y	DDL
ALTER FOREIGN TABLE	Y	Y	DDL
ALTER FUNCTION	Y	Y	DDL
ALTER INDEX	Y	Y	DDL
ALTER LANGUAGE	Y	Y	DDL
ALTER LARGE OBJECT	N	N	N
ALTER MATERIALIZED VIEW	Y	N	N
ALTER OPERATOR	Y	Y	DDL
ALTER OPERATOR CLASS	Y	Y	DDL
ALTER OPERATOR FAMILY	Y	Y	DDL
ALTER PACKAGE	Y	Y	DDL
ALTER POLICY	Y	Y	DDL
ALTER PROCEDURE	Y	Y	DDL
ALTER PROFILE	Y	Y	Details
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

Command	Allowed	Replicated	Lock
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
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

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

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

Command	Allowed	Replicated	Lock
FETCH	Y	N	N
GRANT	Y	Details	DDL
GRANT ROLE	Y	Y	DDL
IMPORT FOREIGN SCHEMA	Y	Y	DDL
LISTEN	Y	N	N
LOAD	Y	N	N
LOAD ROW DATA	Y	Y	DDL
LOCK TABLE	Y	N	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

Command Notes

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 PGD 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.
- `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 PGD 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 PGD
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 PGD
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 PGD 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 PROFILE or ALTER PROFILE

The `PASSWORD_VERIFY_FUNCTION` associated with the profile should be `IMMUTABLE` if the function is `SECURITY DEFINER`. Such a `CREATE PROFILE` or `ALTER PROFILE` command will be replicated but subsequent `CREATE USER` or `ALTER USER` commands using this profile will break the replication due to the `writer` worker throwing the error: `cannot change current role within security-restricted operation`.

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 PGD 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` .

10.6 DDL and role manipulation statements

Users are global objects in a PostgreSQL instance, which means they span multiple databases while PGD operates on an individual database level. Because of this behavior, role manipulation statement handling needs extra thought.

PGD 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.

PGD replicates role manipulation statements if `bdr.role_replication` is enabled (default) and role manipulation statements are run in a PGD-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 PGD-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 PGD apply stalls with an error until the role is created on the other nodes.

> **PGD with non-PGD-enabled databases**
>
> PGD doesn't capture and replicate role management statements when they run on a non-PGD-enabled database in a PGD-enabled PostgreSQL instance. For example, suppose you have databases `bdrdb` (bdr group member) and `postgres` (bare db), and `bdr.role_replication = on`. A `CREATE USER` run in `bdrdb` is replicated, but a `CREATE USER` run in `postgres` isn't.

10.7 Workarounds for DDL restrictions

You can work around some of the limitations of PGD DDL operation handling. Often splitting 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 of 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, PGD 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. PGD 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 later. 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 PGD can execute and results in consistent data across all nodes in a PGD 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

Changing a column's type can cause PostgreSQL to rewrite a table. In some cases, though, you can avoid this rewriting. 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 ..`. Creating ths trigger assigns `NEW.newcolumn` to `NEW.oldcolumn` so that new writes to the table update the new column.

`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` 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 to re-create everything that referred to it.

Changing other types

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

When you use this DDL, 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);
```

10.8 PGD functions that behave like DDL

The following PGD 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 replicated. 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` (deprecated; use `bdr.alter_table_conflict_detection()`)
- `bdr.column_timestamps_disable` (deprecated; use `bdr.alter_table_conflict_detection()`)

Sequence management:

- `bdr.alter_sequence_set_kind`

Stream triggers:

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

11 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.

> **Permissions required**
>
> This means that any user who wants to use sequences must have at least the bdr_application role assigned to them.

PGD global sequences

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

PGD 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 PGD 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 PGD synchronizes inserts between the nodes, they can never conflict.

PGD 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 when the values were created.

SnowflakeId sequences have the restriction that they work only for 64-bit BIGINT datatypes and produce values up to 19 digits long. This might be too long for use in some host language datatypes, such as JavaScript Number 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 PGD global sequence. It can also convert the sequence back to the standard PostgreSQL sequence.

PGD also provides the configuration variable `bdr.default_sequence_kind`. This variable 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` — Newly created sequences are the standard PostgreSQL (local) sequences.
- `galloc` — Always creates globally allocated range sequences.
- `snowflakeid` — 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` — The older version of SnowflakeId sequence. Provided for backward compatibility only. The SnowflakeId is preferred.
- `distributed` (default) — A special value that you can use only for `bdr.default_sequence_kind`. It selects `snowflakeid` for `int8` sequences (that is, `bigserial`) and `galloc` sequence for `int4` (that is, `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're 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 PGD 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, you 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 up to 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, fewer protections, and slower performance.

The differences between timeshard and SnowflakeId are as follows:

- 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 a 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 among the other nodes. This behavior 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 node 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 use—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 might lead to delays or operational issues if the majority of the PGD 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. PGD tracks `galloc` sequences in a special PGD 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. The sequence chunk allocation is managed by Raft, whereas any changes to the sequence name/namespace is managed by the replication stream. So PGD 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 datatype 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.

This 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 help with this use case, PGD lets you 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's the greatest used value and restart the sequence to at least the next value:

```
-- determine highest sequence value across all
nodes
SELECT max((x->'response'->'command_tuples'->0->>'nextval')::bigint)
    FROM
jsonb_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, the sequence starts at `333`, and the cluster has two nodes. The number of allocation is 4, which is 2 per node, and the chunk size is 1000000, which is related to an integer sequence.

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

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

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

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

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

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

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

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

!!! Note

You can't combine it to 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

You can generate globally unique ids in other ways without using the global sequences that can be used with PGD. For example:

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

PGD applications can't use other methods safely. Counter-table-based approaches relying on `SELECT ... FOR UPDATE`, `UPDATE ... RETURNING ...` or similar for sequence generation don't work correctly in PGD because PGD 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 PGD. In most cases, the application coordinates generating sequences that must be gapless from some external source using two-phase commit. Or it generates them only on one node in the PGD group.

UUIDs

`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.

KSUUIDs

PGD provides functions for working with a K-sortable variant of `UUID` data. Known as KSUUID, it 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. This behavior 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.

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

There are two versions of `KSUUID` in PGD: v1 and v2. The legacy `KSUUID` v1 is deprecated but is kept 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.

See KSUUID v2 functions and KSUUID v1 functions in the PGD reference.

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 PGD 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 the 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.

PGD doesn't currently offer any automation for configuring 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)`. 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.

See also

- Global Sequence management interfaces
- KSUUID v2 functions
- KSUUID v1 functions

12 Node management

All data nodes in a PGD cluster are members of one or more groups. By default, all data nodes are members of the top-level group, which spans all data nodes in the PGD cluster. Nodes can also belong to subgroups that can be configured to reflect logical or geographical organization of the PGD cluster.

You can manage nodes and groups using the various options available with nodes and subgroups.

- Creating nodes covers the steps needed to create a new node in a PGD cluster.
- Groups and subgroups goes into more detail on how groups and subgroups work in PGD.
- Creating and joining groups looks at how new PGD groups can be created and how to join PGD nodes to them.
- Viewing topology details commands and SQL queries that can show the structure of a PGD cluster's nodes and groups.
- Removing nodes and groups shows the process to follow to safely remove a node from a group or a group from a cluster.
- Heterogeneous clusters looks at how your PGD cluster can interoperate with PGD nodes from earlier editions of PGD.
- Connection DSNs introduces the DSNs or connection strings needed to connect directly to a node in a PGD cluster. It also covers how to use SSL/TLS certificates to provide authentication and encryption between servers and between clients.
- Replication slots examines how the Postgres replication slots are consumed when PGD is operating.
- Node recovery details the steps needed to bring a node back into service after a failure or scheduled downtime and the impact it has on the cluster as it returns.
- Maintenance commands through proxies shows how to send maintenance commands to nodes that you can't directly access, such as those behind a proxy.

12.1 Creating PGD nodes

Depending on your selected deployment method, you may or may not have to create PGD nodes manually. For example, if you are using TPA or the EDB Postgres Distributed for Kubernetes, the nodes are created automatically. But if you are manually deploying PGD or creating your own deployment method, you need to know how to create and configure a PGD node.

It's just Postgres

A PGD node is just a Postgres instance with the BDR extension installed. The BDR extension enables bidirectional replication between nodes and is the foundation of PGD.

That means, in the most general terms, you can create a PGD node by installing Postgres and the BDR extension, and then configuring the node to connect to the other nodes in the PGD group. But there are some specifics to consider.

Which Postgres version?

PGD is built on top of Postgres, so the distribution and version of Postgres you use for your PGD nodes is important. The version of Postgres you use must be compatible with the version of PGD you are using. You can find the compatibility matrix in the release notes. Features and functionality in PGD may depend on the distribution of Postgres you are using. The EDB Postgres Advanced Server is the recommended distribution for PGD. PGD also supports EDB Postgres Extended Server and Community Postgres. You can find out what features are available in each distribution in the Planning section's Choosing a server page.

Installing Postgres

You must install your selected Postgres distribution on each node you are configuring. You can find installation instructions for each distribution in the EDB Postgres Advanced Server documentation, EDB Postgres Extended Server documentation, and the Postgres installation documentation. You can also refer to the PGD manual installation guide which covers the installation of Postgres.

Installing the BDR extension

The BDR extension is the key to PGD's distributed architecture. You need to install the BDR extension on each node in your PGD cluster. The BDR extension is available from the EDB Postgres Distributed repository. You need to add the `postgres_distributed` repository to your package management system on Linux and then install the `edb-bdr` package. You can find the repository configuration instructions in the PGD manual installation guide.

Once the repository is configured, you can install the BDR package with your package manager. The package name is `edb-bdr5-<postgresversion>` where `<postgresversion>` is the version of Postgres you are using. For example, if you are using Postgres 13, the package name is `edb-bdr5-13`.

Configuring the database for PGD

This process is specific to PGD and involves configuring the Postgres instance to work with the BDR extension and adjusting various settings to work with the PGD cluster. This process is also detailed in the PGD manual installation guide The steps are as follows:

- Add the BDR extension `$libdir/bdr` at the start of the `shared_preload_libraries` setting in `postgresql.conf`.
- Set the `wal_level` GUC variable to `logical` in `postgresql.conf`.
- Turn on commit timestamp tracking by setting `track_commit_timestamp` to `'on'` in `postgresql.conf`.

- Increase the maximum worker processes to 16 or higher by setting `max_worker_processes` to `'16'` in `postgresql.conf`.

> **The `max_worker_processes` value**
>
> The `max_worker_processes` value is derived from the topology of the cluster, the number of peers, number of databases, and other factors. To calculate the needed value, see Postgres configuration/settings. The value of 16 was calculated for the size of cluster being deployed in this example. It must be increased for larger clusters.

- Set a password on the EnterprisedDB/Postgres user.
- Add rules to `pg_hba.conf` to allow nodes to connect to each other.
    - Ensure that these lines are present in `pg_hba.conf`:

```
host all all all md5
host replication all all md5
```

- Add a `.pgpass` file to allow nodes to authenticate each other.
    - Configure a user with sufficient privileges to log in to the other nodes.
    - See The Password File in the Postgres documentation for more on the `.pgpass` file.

Once these steps are complete, restart the Postgres instance to apply the changes.

Initializing a PGD node

Log into the database instance you have configured and set up the BDR extension. You can do this by running the `CREATE EXTENSION bdr;` command as super user in the database. This command creates the BDR extension.

You also need to create a database within Postgres to use as PGD's replicated database. You can do this with the `CREATE DATABASE` command. The created database should be the name of the database that other nodes in the PGD cluster replicate. The convention is to name the database `bdrdb`.

Next steps

The node is now configured and ready to be join a group, or start a group, in the PGD cluster.

12.2 Groups and subgroups

Groups

A PGD cluster's nodes are gathered in groups. A "top level" group always exists and is the group to which all data nodes belong to automatically. The "top level" group can also be the direct parent of sub-groups.

Sub-groups

A group can also contain zero or more subgroups. Subgroups can be used to represent data centers or locations allowing commit scopes to refer to nodes in a particular region as a whole. PGD Proxy can also make use of subgroups to delineate nodes available to be write leader.

The `node_group_type` value specifies the type when the subgroup is created. Some sub-group types change the behavior of the nodes within the group. For example, a subscriber-only sub-group will make all the nodes within the group into subscriber-only nodes.

12.3 Creating and joining PGD groups

Creating and joining PGD groups

For PGD, 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 PGD node created, must know the PostgreSQL connection string that other nodes can use to connect to it. This connection string is sometimes referred to as a data source name (DSN).

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 PGD group from the local node. Doing so activates PGD on that node and allows other nodes to join the PGD 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 PGD 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`. 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 PGD node is joined to an existing PGD 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 PGD 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's joining the cluster must not contain any schema or data that already exists on databases in the PGD 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. Also, if a non-superuser is performing the joining operation, extensions that require superuser permission must be created manually. For more details, see Connections and roles.

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 (logically close, ideally with low latency and high bandwidth) 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. These states 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 approach 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's 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.

12.4 Viewing PGD topology

Listing PGD groups

Using pgd-cli

Use the pgd-cli `show-groups` command to list all groups in the PGD cluster:

```
pgd show-groups
```

```
  Group    Group ID   Type            Parent Group Location Raft  Routing Write Leader
  -----    --------   ----            ------------ -------- ----  ------- ------------
  bdrgroup 1360502012 global                                true  false
  group_a  3618712053 data            bdrgroup     a        true  true    bdr-a1
  group_b  402614658  data            bdrgroup     b        true  true    bdr-b1
  group_c  2808307099 data            bdrgroup     c        false false
  group_so 2123208041 subscriber-only bdrgroup     c        false false
```

Using SQL

The following simple query lists all the PGD node groups of which the current node is a member. It currently returns only one row from `bdr.local_node_summary`.

```
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 PGD group

Using pgd-cli

Use the `show-nodes` command to list all nodes in the PGD cluster:

```
pgd show-nodes
```

```
 Node               Node ID    Group    Type            Current State Target State Status Seq ID
 ----               -------    -----    ----            ------------- ------------ ------ ------
 bdr-a1             3136956818 group_a  data            ACTIVE        ACTIVE       Up     6
 bdr-a2             2133699692 group_a  data            ACTIVE        ACTIVE       Up     3
 logical-standby-a1 1140256918 group_a  standby         STANDBY       STANDBY      Up     9
 witness-a          3889635963 group_a  witness         ACTIVE        ACTIVE       Up     7
 bdr-b1             2380210996 group_b  data            ACTIVE        ACTIVE       Up     1
 bdr-b2             2244996162 group_b  data            ACTIVE        ACTIVE       Up     2
 logical-standby-b1 3541792022 group_b  standby         STANDBY       STANDBY      Up     10
 witness-b          661050297  group_b  witness         ACTIVE        ACTIVE       Up     5
 witness-c          1954444188 group_c  witness         ACTIVE        ACTIVE       Up     4
 subscriber-only-c1 2448841809 group_so subscriber-only ACTIVE        ACTIVE       Up     8
```

Use `grep` with the group name to filter the list to a specific group:

```
pgd show-nodes | grep group_b
```

```
 bdr-b1             2380210996 group_b  data            ACTIVE        ACTIVE       Up     1
 bdr-b2             2244996162 group_b  data            ACTIVE        ACTIVE       Up     2
 logical-standby-b1 3541792022 group_b  standby         STANDBY       STANDBY      Up     10
 witness-b          661050297  group_b  witness         ACTIVE        ACTIVE       Up     5
```

Using SQL

You can extract the list of all nodes in a given node group (such as `mygroup`) from the `bdr.node_summary` view. For 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`.

12.5 Removing nodes and groups

Removing a node from a PGD group

Since PGD 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 PGD 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 PGD but doesn'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()`.

Removing a whole PGD group

PGD groups usually map to locations. When a location is no longer being deployed, it's likely that the PGD group for the location also needs to be removed.

The PGD group that's being removed must be empty. Before you can remove the group, you must part all the nodes in the group.

12.6 Joining a heterogeneous cluster

A PGD 4.0 node can join an 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 PGD 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 an 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.

PGD 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. However, 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. Using this method 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.

12.7 Connection DSNs and SSL (TLS)

Because nodes connect using `libpq`, the DSN of a node is a `libpq` connection string. As such, the connection string 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's 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 PGD connection, which 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.

12.8 Replication slots created by PGD

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

- 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 PGD nodes in the cluster, including direct logical standbys, if any

> **Warning**
>
> Don't drop those slots. PGD 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 PGD. 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 PGD is used to replicate the `mydb` database and the PGD 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 PGD primarily to track the oldest safe position that any node in the PGD 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 PGD group without having all existing nodes up and running (although the majority of nodes should be up). This process doesn't incur 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. PGD handles this by shortening the database name, the PGD 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 PGD group named `group20xxxxxxxxxxxxx` (20 bytes long). The logical replication slot associated to node `a30xxxxxxxxxxxxxxxxxxxxxxxxxxxx` (30 bytes long) is called since `3597186`, `be9cbd0`, and `7f304a2` are respectively the hashes of `db20xxxxxxxxxxxxxxx`, `group20xxxxxxxxxxxxx`, and `a30xxxxxxxxxxxxxxxxxxxxxxxxxxxx`.

```
bdr_db20xxxx3597186_group20xbe9cbd0_a30xxxxxxxxxxxxx7f304a2
```

12.9 Node restart and down node recovery

PGD 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 up in `bdr.node_slots` with `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 PGD. 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 PGD.

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.)

12.10 Maintenance commands through proxies

Maintenance and performance

As a general rule, you should never perform maintenance operations on a cluster's write leader. Maintenance operations such as `VACUUM` can be quite disruptive to the smooth running of a busy server and often detrimental to workload performance. Therefore, it's best to run maintenance commands on any node in a group that isn't the write leader. Generally, this requires you to connect directly and issue the maintenance commands on the non-write-leader nodes. But in some situations, this isn't possible.

Maintenance and proxies

Proxies, by design, always connect to and send commands to the current write leader. This usually means that you must not connect by way of a proxy to perform maintenance. PGD clusters nodes can present a direct connection for psql and PGD CLI clients that you can use to issue maintenance commands to the server on those nodes. But there are environment in which the PGD cluster is deployed where a proxy is the only way to access the cluster.

For example, in BigAnimal, PGD clusters are locked down such that the only access to the database is through an instance of PGD Proxy. This configuration reduces the footprint of the cluster and makes it more secure. However, it requires that you use a different way of sending maintenance requests to the cluster's nodes.

The technique outlined here is generally useful for despatching commands to specific nodes without being directly connected to that node's server.

Maintenance commands

The term *maintenance commands* refers to:

- `VACUUM`
- Non-replicated DDL commands (which you might want to manually replicate)

A note on node names

The servers in the cluster are referred to by their PGD cluster node names. To get a list of node names in your cluster, use:

```
select node_name from bdr.node;
```

> **Tip**
>
> For more details, see the `bdr.node` table.

This command lists just the node names. If you need to know the group they are a member of, use:

```
select node_name, node_group_name from bdr.node_summary;
```

> **Tip**
>
> For more details, see the `bdr.node_summary` table.

Finding the write leader

If you're connected through the proxy, then you're connected to the write leader. Run `select node_name from bdr.local_node_summary` to see the name of the node:

```
select node_name from bdr.local_node_summary;
```

output

```
    node_name
------------------
node-two
(1 row)
```

This is the node you do **not** want to run your maintenance tasks on.

```
select * from bdr.node_group_routing_summary;
```

output

```
 node_group_name |     write_lead     | previous_write_lead |       read_nodes
-----------------+-------------------+---------------------+-------------------------
 dc1             | node-two          | node-one            | {node-one,node-three}
```

Where the `write_lead` is the node determined earlier (node-two), you can also see the two `read_nodes` (node-one and node-three). It's on these nodes that you can safely perform maintenance.

> **Tip**
>
> You can perform that operation with a single query:
>
> ```
> select read_nodes from bdr.node_group_routing_summary where write_lead = (select node_name from
> bdr.local_node_summary);
> ```

Using `bdr.run_on_nodes()`

PGD has the ability to run specific commands on specific nodes using the `bdr.run_on_nodes()` function. This function takes two parameters: an array of node names and the command you want to run on those nodes. For example:

```
SELECT bdr.run_on_nodes(ARRAY['node-one','node-three'],'vacuum full foo');
```

output

```
                                                                  run_on_nodes
----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
 [{"dsn": "host=host-one port=5444 dbname=bdrdb", "node_id": "807899305", "response": {"command_status": "VACUUM"}, "node_name": "node-one", "query_send_time": "2024-01-16 16:24:35.418323+00"}, {"dsn": "host=host-three port=5432 dbname=bdrdb", "node_id": "199017004", "response": {"command_status": "VACUUM"}, "node_name": "node", "query_send_time": "2024-01-16 16:24:35.4542+00"}]
```

This command runs the `vacuum full foo` command on the node-one and node-three nodes. The node names are passed to the function in an array.

The `bdr.run_on_nodes` function reports its results as JSONB. The results include the name of the node and the response (or error message) resulting from running the command. Other fields included might be include and might not be relevant.

The results also appear as a single string that's hard to read. By applying some formatting to this string, it can become more readable.

Formatting `bdr.run_on_nodes()` output

Using Postgres's JSON expressions, you can reduce the output to just the columns you're interested in. The following command is functionally equivalent to the previous example but lists only the node and response as its results:

```
select q->>'node_name' as node, q->>'response' as response FROM
jsonb_array_elements(bdr.run_on_nodes(ARRAY['node-one','node-three'], 'VACUUM FULL foo')) q;
```

```
                                                output
         node     |         response
------------------+---------------------------
 node-one         | {"command_status": "VACUUM"}
 node-three       | {"command_status": "VACUUM"}
```

If an error occurs, the `command_status` field is set to error. An additional `error_message` value is included in the response. For example:

```
select q->>'node_name' as node, q->>'response' as response FROM
jsonb_array_elements(bdr.run_on_nodes(ARRAY['node-one','node-three'], 'VACUUM FULL fool')) q;
```

```
                                                output
    node          |                                          response
------------------+-------------------------------------------------------------------------------------------
 node-one         | {"error_message": "ERROR:  relation \"fool\" does not exist\n", "command_status": "ERROR"}
 node-three       | {"error_message": "ERROR:  relation \"fool\" does not exist\n", "command_status": "ERROR"}
(2 rows)
```

Defining a function for maintenance

If you find yourself regularly issuing maintenance commands to one node at a time, you can define a function to simplify things:

```
create or replace function runmaint(nodename varchar, command varchar) returns TABLE(node text,response jsonb) as
$$

begin
return query
select (q->>'node_name')::text, (q->'response') from jsonb_array_elements(bdr.run_on_nodes(ARRAY [nodename],
command)) as q;
end;
$$ language
'plpgsql';
```

This function takes a node name and a command and runs the command on that node, returning the results as shown in this interaction:

```
select runmaint('node-one','VACUUM FULL foo');
```

```
                                                output
                     runmaint
-----------------------------------------------------
 (node-one,"{""command_status"": ""VACUUM""}")
```

You can break up the response by using `select * from`:

```
select * from runmaint('node-one','VACUUM FULL foo');
```

output

```
    node          |         response
------------------+-------------------------------
 node-one         | {"command_status": "VACUUM"}
(1 row)
```

13 Postgres configuration

Several Postgres configuration parameters affect PGD nodes. You can set these parameters differently on each node, although we don't generally recommend it.

For PGD's own settings, see the PGD settings reference.

Postgres settings

To run correctly, PGD requires these Postgres settings:

- `wal_level` — Must be set to `logical`, since PGD relies on logical decoding.
- `shared_preload_libraries` — Must include `bdr` to enable the extension. Most other extensions can appear before or after the `bdr` entry in the comma-separated list. One exception to that is `pgaudit`, which must appear in the list before `bdr`. Also, don't include `pglogical` in this list.
- `track_commit_timestamp` — Must be set to `on` for conflict resolution to retrieve the timestamp for each conflicting row.

PGD 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 size overflow the buffer and are stored temporarily on local disk. Default is 64MB, but you can set it much higher.
- `max_worker_processes` — PGD 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 to add together these values:
  - One per PostgreSQL instance
  - One per database on that instance
  - Four per PGD-enabled database
  - One per peer node in the PGD group
  - The number of peer nodes times the (number of writers (bdr.num_writers) plus one) You might need more worker processes temporarily when a node is being removed from a PGD group.

- `max_wal_senders` — Two needed for every peer node.
- `max_replication_slots` — Two needed for every peer node.
- `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, PGD requires N slots and WAL senders. During synchronization, PGD 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 `max_replication_slots` also sets the maximum number of replication origins, and some of the functionality of Parallel Apply uses an extra origin per writer.

When the decoding worker is enabled, this process requires one extra replication slot per PGD group.

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

A legacy synchronous replication mode is supported using the following parameters. See Commit scopes for details and limitations.

- `synchronous_commit` and `synchronous_standby_names` — Affects the durability and performance of PGD replication. in a similar way to physical replication.

Time-based snapshots

`snapshot_timestamp`

Turns on the use of timestamp-based snapshots and sets the timestamp to use.

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. This parameter can be set only at Postgres server start.

14 PGD Proxy

Managing application connections is an important part of high availability. PGD Proxy offers a way to manage connections to the EDB Postgres Distributed cluster. It acts as a proxy layer between the client application and the Postgres database.

- PGD Proxy overview provides an overview of the PGD Proxy, its processes, and how it interacts with the EDB Postgres Distributed cluster.
- Installing the PGD Proxy service covers installation of the PGD Proxy service on a host.
- Configuring PGD Proxy details the three levels (group, node, and proxy) of configuration on a cluster that control how the PGD Proxy service behaves.
- Administering PGD Proxy shows how to switch the write leader and manage the PGD Proxy.
- Monitoring PGD Proxy looks at how to monitor PGD Proxy through the cluster and at a service level.
- Read-only routing explains how the read-only routing feature in PGD Proxy enables read scalability.
- Raft provides an overview of the Raft consensus mechanism used to coordinate PGD Proxy.

14.1 EDB Postgres Distributed Proxy overview

Especially with asynchronous replication, having a consistent write leader node is important to avoid conflicts and guarantee availability for the application.

The two parts to EDB Postgres Distributed's proxy layer are:

- Proxy configuration and routing information, which is maintained by the PGD consensus mechanism.
- The PGD Proxy service, which is installed on a host. It connects to the PGD cluster, where it reads its configuration and listens for changes to the routing information.

This layer is normally installed in a highly available configuration (at least two instances of the proxy service per PGD group).

Once configured, the PGD Proxy service monitors routing changes as decided by the EDB Postgres Distributed cluster. It acts on these changes to ensure that connections are consistently routed to the correct nodes.

Configuration changes to the PGD Proxy service are made through the PGD cluster. The PGD Proxy service reads its configuration from the PGD cluster, but the proxy service must be restarted to apply those changes.

The information about currently selected write and read nodes is visible in `bdr.node_group_routing_summary`. This is a node-local view: the proxy always reads from Raft leader to get a current and consistent view.

Leader selection

The write leader is selected by the current Raft leader for the group the proxy is part of. This could be the Raft leader for a subgroup or leader for the entire cluster. The leader is selected from candidate nodes that are reachable and meet the criteria based on the configuration as described in PGD Proxy cluster configuration. To be a viable candidate, the node must have `route_writes` enabled and `route_fence` disabled and be within `route_writer_max_lag` (if enabled) from the previous leader. The candidates are ordered by their `route_priority` in descending order and by the lag from the previous leader in ascending order.

The new leader selection process is started either when there's no existing leader currently or when connectivity is lost to the existing leader. (If there's no existing write leader, it could be because there were no valid candidates or because Raft was down.)

A node is considered connected if the last Raft protocol message received from the leader isn't older than Raft election timeout (see Internal settings - Raft timeouts).

Since the Raft leader is sending heartbeat 3 times every election timeout limit, the leader node needs to miss the reply to 3 heartbeats before it's considered disconnected.

PGD Proxy cluster configuration

The PGD cluster always has at least one top-level group and one data group. PGD elects the write leader for each data group that has the `enable_proxy_routing` and `enable_raft` options set to true.

The cluster also maintains proxy configurations for each group. Each configuration has a name and is associated with a group. You can attach a proxy to a top-level group or data group. You can attach multiple proxies to each group. When a PGD Proxy service starts running on a host, it has a name in its local configuration file and it connects to a node in a group. From there, it uses the name to look up its complete configuration as stored on the group.

PGD Proxy service

The EDB Postgres Distributed Proxy (PGD Proxy) service is a process that acts as an abstraction layer between the client application and Postgres. It interfaces with the PGD consensus mechanism to get the identity of the current write leader node and redirects traffic to that node. It also optionally supports a read-only mode where it can route read-only queries to nodes that aren't the write leader, improving the overall performance of the cluster.

PGD Proxy is a TCP layer 4 proxy.

How they work together

Upon starting, PGD Proxy connects to one of the endpoints given in the local config file. It fetches:

- DB connection information for all nodes.
- Proxy options like listen address, listen port.
- Routing details including the current write leader in default mode, read nodes in read-only mode, or both in any mode.

The endpoints given in the config file are used only at startup. After that, actual endpoints are taken from the PGD catalog's `route_dsn` field in `bdr.node_routing_config_summary`.

PGD manages write leader election. PGD Proxy interacts with PGD to get write leader change events notifications on Postgres notify/listen channels and routes client traffic to the current write leader. PGD Proxy disconnects all existing client connections on write leader change or when write leader is unavailable. Write leader election is a Raft-backed activity and is subject to Raft leader availability. PGD Proxy closes the new client connections if the write leader is unavailable.

PGD Proxy responds to write leader change events that can be categorized into two modes of operation: *failover* and *switchover*.

Automatic transfer of write leadership from the current write leader node to a new node in the event of Postgres or operating system crash is called *failover*. PGD elects a new write leader when the current write leader goes down or becomes unresponsive. Once the new write leader is elected by PGD, PGD Proxy closes existing client connections to the old write leader and redirects new client connections to the newly elected write leader.

User-controlled, manual transfer of write leadership from the current write leader to a new target leader is called *switchover*. Switchover is triggered through the PGD CLI group set leader command. The command is submitted to PGD, which attempts to elect the given target node as the new write leader. Similar to failover, PGD Proxy closes existing client connections and redirects new client connections to the newly elected write leader. This is useful during server maintenance, for example, if the current write leader node needs to be stopped for maintenance like a server update or OS patch update.

If the proxy is configured to support read-only routing, it can route read-only queries to a pool of nodes that aren't the write leader. The pool of nodes is maintained by the PGD cluster and proxies listen for changes to the pool. When the pool changes, the proxy updates its routing configuration and starts routing read-only queries to the new pool of nodes and disconnecting existing client connections to nodes that have left the pool.

Consensus grace period

PGD Proxy provides the `consensus_grace_period` proxy option that can be used to configure the routing behavior upon loss of a Raft leader. PGD Proxy continues to route to the current write leader (if it's available) for this duration. If the new Raft leader isn't elected during this period, the proxy stops routing. If set to `0s`, PGD Proxy stops routing immediately.

The main purpose of this option is to allow users to configure the write behavior when the Raft leader is lost. When the Raft leader isn't present in the cluster, it's not always guaranteed that the current write leader seen by the proxy is the correct one. In some cases, like network partition in the following example, it's possible that the two write leaders may be seen by two different proxies attached to the same group, increasing the chances of write conflicts. If this isn't the behavior you want, then you can set the previously mentioned `consensus_grace_period` to 0s. This setting configures the proxy to stop routing and closes existing open connections immediately when it detects the Raft leader is lost.

Network partition example

Consider a 3-data-node group with a proxy on each data node. In this case, if the current write leader gets network partitioned or isolated, then the data nodes present in the majority partition elect a new write leader. If `consensus_grace_period` is set to a non-zero value, for example, `10s`, then the proxy present on the previous write leader continues to route writes for this duration.

In this case, if the grace period is kept too high, then writes continue to happen on the two write leaders. This condition increases the chances of write conflicts.

Having said that, most of the time, upon loss of the current Raft leader, the new Raft leader gets elected by BDR within a few seconds if more than half of the nodes (quorum) are still up. Hence, if the Raft leader is down but the write leader is still up, then proxy can be configured to allow routing by keeping `consensus_grace_period` to a non-zero, positive value. The proxy waits for the Raft leader to get elected during this period before stopping routing. This might be helpful in some cases where availability is more important.

Read consensus grace period

Similar to the `consensus_grace_period`, a `read_consensus_grace_period` option is available for read-only routing. This option can be used to configure the routing behavior upon loss of a Raft leader for read-only queries. PGD Proxy continues to route to the current read nodes for this duration. If the new Raft leader isn't elected during this period, the proxy stops routing read-only queries. If set to `0s`, PGD Proxy stops routing read-only queries immediately.

Multi-host connection strings

The PostgreSQL C client library (libpq) allows you to specify multiple host names in a single connection string for simple failover. This ability is also supported by client libraries (drivers) in some other programming languages. It works well for failing over across PGD Proxy instances that are down or inaccessible.

If an application connects to a proxy instance that doesn't have access to a write leader, the connection will simply fail. No other hosts in the multi-host connection string will be tried. This behavior is consistent with the behavior of PostgreSQL client libraries with other proxies like HAProxy or pgbouncer. Access to a write leader requires the group the instance is part of has been able to select a write leader for the group.

14.2 Installing PGD Proxy

Installing PGD Proxy

You can use two methods to install and configure PGD Proxy to manage an EDB Postgres Distributed cluster. The recommended way to install and configure PGD Proxy is to use the EDB Trusted Postgres Architect (TPA) utility for cluster deployment and management.

Installing through TPA

If the PGD cluster is being deployed through TPA, then TPA installs and configures PGD Proxy automatically as per the recommended architecture. If you want to install PGD Proxy on any other node in a PGD cluster, then you need to attach the pgd-proxy role to that instance in the TPA configuration file. Also set the `bdr_child_group` parameter before deploying, as this example shows. See Trusted Postgres Architect for more information.

```
- Name: proxy-
a1
  location:
a
  node: 4
  role:
  - pgd-proxy
  vars:
    bdr_child_group: group_a
  volumes:
  - device_name:
/dev/sdf
    volume_type: none
```

Configuration

PGD Proxy connects to the PGD database for its internal operations, like getting proxy options and getting write leader details. Therefore, it needs a list of endpoints/dsn to connect to PGD nodes. PGD Proxy expects these configurations in a local config file `pgd-proxy-config.yml`. Following is a working example of the `pgd-proxy-config.yml` file:

```
log-level: debug
cluster:
  name: cluster-
name
  endpoints:
    - "host=bdr-a1 port=5432 dbname=bdrdb
user=pgdproxy"
    - "host=bdr-a3 port=5432 dbname=bdrdb
user=pgdproxy"
    - "host=bdr-a2 port=5432 dbname=bdrdb
user=pgdproxy"
  proxy:
    name: "proxy-a1"
```

By default, in the cluster created through TPA, `pgd-proxy-config.yml` is located in the `/etc/edb/pgd-proxy` directory. PGD Proxy searches for `pgd-proxy-config.yml` in the following locations. Precedence order is high to low.

1. `/etc/edb/pgd-proxy` (default)
2. `$HOME/.edb/pgd-proxy`

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 starting a service. See the sample service file.

You can set the log level for the PGD Proxy service using the top-level config parameter `log-level`, as shown in the sample config. The valid values for `log-level` are `debug`, `info`, `warn`, and `error`.

`cluster.endpoints` and `cluster.proxy.name` are mandatory fields in the config file. PGD Proxy always tries to connect to the first endpoint in the list. If it fails, it tries the next endpoint, and so on.

PGD Proxy uses endpoints given in the local config file only at proxy startup. After that, PGD Proxy retrieves the list of actual endpoints (route_dsn) from the PGD Proxy catalog. Therefore, the node option `route_dsn` must be set for each PGD Proxy node. See route_dsn for more information.

Configuring health check

PGD Proxy provides HTTP(S) health check APIs. If the health checks are required, you can enable them by adding the following configuration parameters to the pgd-proxy configuration file. By default, it's disabled.

```
cluster:
  name: cluster-
name
  endpoints:
    - "host=bdr-a1 port=5432 dbname=bdrdb user=pgdproxy
"
    - "host=bdr-a3 port=5432 dbname=bdrdb user=pgdproxy
"
    - "host=bdr-a2 port=5432 dbname=bdrdb user=pgdproxy
"
  proxy:
    name: "proxy-a1"
    endpoint: "host=proxy-a1 port=6432 dbname=bdrdb user=pgdproxy
"
    http:
      enable: true
      host: "0.0.0.0"
      port: 8080
      secure: false
      cert_file: ""
      key_file: ""
      probes:
        timeout:
10s
```

You can enable the API by adding the config `cluster.proxy.http.enable: true`. When enabled, an HTTP server listens on the default port, `8080`, with a 10-second `timeout` and no HTTPS support.

To enable HTTPS, set the config parameter `cluster.proxy.http.secure: true`. If it's set to `true`, you must also set the `cert_file` and `key_file`.

The `cluster.proxy.endpoint` is an endpoint used by the proxy to connect to the current write leader as part of its checks. When `cluster.proxy.http.enable` is `true`, `cluster.proxy.endpoint` must also be set. It can be the same as BDR noderouting_dsn, where host is `listen_address` and port is `listen_port` proxy options. If required, you can add connection string parameters in this endpoint, like `sslmode`, `sslrootcert`, `user`, and so on.

PGD Proxy user

The database user specified in the endpoint doesn't need to be a superuser. Typically, in the TPA environment, pgdproxy is an OS user as well as a database user with the bdr_superuser role.

PGD Proxy service

We recommend running PGD Proxy as a systemd service. The `pgd-proxy` service unit file is located at `/etc/systemd/system/pgd-proxy.service` by default. Following is the sample service file created by TPA:

```
[Unit]
Description=PGD Proxy

[Service]
Type=simple
User=pgdproxy
Group=pgdproxy
Restart=on-failure
RestartSec=1s
ExecStart=/usr/bin/pgd-proxy -f /etc/edb/pgd-proxy/pgd-proxy-config.yml
StandardOutput=syslog
StandardError=syslog
SyslogIdentifier=pgd-proxy

[Install]
WantedBy=multi-user.target
```

Use these commands to manage the `pgd-proxy` service:

```
systemctl status pgd-
proxy
systemctl stop pgd-proxy
systemctl restart pgd-proxy
```

Installing manually

You can manually install PGD Proxy on any Linux machine using `.deb` and `.rpm` packages available from the PGD repository. The package name is `edb-pgd5-proxy`. For example:

```
# for
Debian
sudo apt-get install edb-pgd5-proxy
```

14.3 PGD Proxy configuration

Group-level configuration

Configuring the routing is done either through SQL interfaces or through PGD CLI.

You can enable routing decisions by calling the `bdr.alter_node_group_option()` function. For example:

```
SELECT bdr.alter_node_group_option('region1-group', 'enable_proxy_routing', 'true')
```

You can disable it by setting the same option to `false`.

Additional group-level options affect the routing decisions:

- `route_writer_max_lag` — Maximum lag in bytes of the new write candidate to be selected as write leader. If no candidate passes this, no writer is selected automatically.
- `route_reader_max_lag` — Maximum lag in bytes for a node to be considered a viable read-only node (PGD 5.5.0 and later).

Node-level configuration

Set per-node configuration of routing using `bdr.alter_node_option()`. The available options that affect routing are:

- `route_dsn` — The dsn used by proxy to connect to this node.
- `route_priority` — Relative routing priority of the node against other nodes in the same node group. Used only when electing a write leader.
- `route_fence` — Determines whether the node is fenced from routing. When fenced, the node can't receive connections from PGD Proxy. It therefore can't become the write leader or be available in the read-only node pool.
- `route_writes` — Determines whether writes can be routed to this node, that is, whether the node can become write leader.
- `route_reads` — Determines whether read-only connections can be routed to this node (PGD 5.5.0 and later).

Proxy-level configuration

You can configure the proxies using SQL interfaces.

Creating and dropping proxy configurations

You can add a proxy configuration using `bdr.create_proxy`. For example, `SELECT bdr.create_proxy('region1-proxy1', 'region1-group');` creates the default configuration for a proxy named `region1-proxy1` in the PGD group `region1-group`.

The name of the proxy given here must be same as the name given in the proxy configuration file.

You can remove a proxy configuration using `SELECT bdr.drop_proxy('region1-proxy1')`. Dropping a proxy deactivates it.

Altering proxy configurations

You can configure options for each proxy using the `bdr.alter_proxy_option()` function.

The available options are:

- `listen_address` — Address for the proxy to listen on.
- `listen_port` — Port for the proxy to listen on.
- `max_client_conn` — Maximum number of connections for the proxy to accept.
- `max_server_conn` — Maximum number of connections the proxy can make to the Postgres node.
- `server_conn_timeout` — Connection timeout for server connections.
- `server_conn_keepalive` — Keepalive interval for server connections.
- `consensus_grace_period` — Duration for which proxy continues to route even upon loss of a Raft leader. If set to `0s`, proxy stops routing immediately.
- `read_listen_address` — Address for the read-only proxy to listen on.
- `read_listen_port` — Port for the read-only proxy to listen on.
- `read_max_client_conn` — Maximum number of connections for the read-only proxy to accept.
- `read_max_server_conn` — Maximum number of connections the read-only proxy can make to the Postgres node.
- `read_server_conn_keepalive` — Keepalive interval for read-only server connections.
- `read_server_conn_timeout` — Connection timeout for read-only server connections.
- `read_consensus_grace_period` — Duration for which read-only proxy continues to route even upon loss of a Raft leader.

14.4 Administering PGD Proxy

Switching the write leader

Switching the write leader is a manual operation that you can perform to change the node that's the write leader. It can be useful when you want to perform maintenance on the current write leader node or when you want to change the write leader for any other reason. When changing write leader, there are two modes: `strict` and `fast`. In `strict` mode, the lag is checked before switching the write leader. It waits until the lag is less than `route_writer_max_lag` before starting the switchover. This is the default. In `fast` mode, the write leader is switched immediately. You can also set a timeout parameter to specify the time to wait for the switchover to complete.

> **Note**
>
> The switchover operation is not a guaranteed operation. If, due to a timeout or for other reasons, the switchover to the given target node fails, PGD may elect another node as write leader in its place. This other node can include the current write leader node. PGD always tries to elect a new write leader if the switchover operation fails.

Using SQL

You can perform a switchover operation that explicitly changes the node that's the write leader to another node.

Use the `bdr.routing_leadership_transfer()` function.

For example, to switch the write leader to node `node1` in group `group1`, use the following SQL command:

```
SELECT bdr.routing_leadership_transfer('group1', 'node1','strict','10s');
```

This command switches the write leader using `strict` mode and waits for up to 10 seconds for the switchover to complete. Those are default settings, so you can omit them, as follows:

```
SELECT bdr.routing_leadership_transfer('group1', 'node1');
```

Using PGD CLI

You can use the `switchover` command to perform a switchover operation.

For example, to switch the write leader from node `node1` to node `node2` in group `group1`, use the following command:

```
pgd switchover --node-group group1 --node-name node1 --method strict --timeout
10s
```

This command switches the write leader using `strict` mode and waits for up to 10 seconds for the switchover to complete. Those are default settings, so you can omit them, as follows:

```
pgd switchover --node-group group1 --node-name
node1
```

14.5 Monitoring PGD Proxy

You cam monitor proxies at the cluster and group level or at the process level.

Monitoring through the cluster

Using SQL

The current configuration of every group is visible in the `bdr.node_group_routing_config_summary` view.

The `bdr.node_routing_config_summary` view shows current per-node routing configuration.

`bdr.proxy_config_summary` shows per-proxy configuration.

Monitoring at the process level

Proxy health check

PGD Proxy provides the following HTTP(S) health check API endpoints. The API endpoints respond to `GET` requests. You need to enable and configure the endpoints before using them. See Configuration.

Endpoint	Description
`/health/is-ready`	Checks if the proxy can successfully route connections to the current write leader.
`/health/is-live`	Checks if the proxy is running.
`/health/is-write-ready`	Checks if the proxy can successfully route connections to the current write leader (PGD 5.5.0 and later).
`/health/is-read-only-ready`	Checks if the proxy can successfully route read-only connections (PGD 5.5.0 and later).

Readiness

On receiving a valid `GET` request:

- When in default (write) mode, the proxy checks if it can successfully route connections to the current write leader.
- When in read-only mode, the proxy checks if it can successfully route read-only connections.
- When in any mode, the proxy first checks if it can successfully route connections to the current write leader. If it can, the check is successful. If not, it checks if it can route a read-only connection. If it can, the check is successful. If not, the check fails.

If the check returns successfully, the API responds with a body containing `true` and an HTTP status code `200 (OK)`. Otherwise, it returns a body containing `false` with the HTTP status code `500 (Internal Server Error)`.

Liveness

Liveness checks return either `true` with HTTP status code `200 (OK)` or an error. They never return `false` because the HTTP server listening for the request is stopped if the PGD Proxy service fails to start or exits.

Proxy log location

Proxies also write logs to system logging where they can be monitored with other system services.

syslog

- Debian based - `/var/log/syslog`
- Red Hat based - `/var/log/messages`

Use the `journalctl` command to filter and view logs for troubleshooting PGD Proxy. The following are sample commands for quick reference:

```
journalctl -u pgd-proxy -n100 -f
journalctl -u pgd-proxy --since today
journalctl -u pgd-proxy --since "10 min
ago"
journalctl -u pgd-proxy --since "2022-10-20 16:21:50" --until "2022-10-20 16:21:55"
```

14.6 Read-only routing with PGD Proxy

Background

By default, PGD Proxy routes connections to the currently selected write leader in the cluster. This allows the write traffic conflicts to be rapidly and consistently resolved. Just routing everything to a single node, the write leader, is a natural fit for traditional high-availability deployments where system throughput is typically limited to the throughput of what a single node can handle.

But for some use cases, this behavior also means that clients that are only querying the data are also placing a load on the current write leader. It's possible this read-only workload could be equally well served by one of the non-write-leader nodes in the cluster.

If you could move traffic that has read-only queries to the non-write leader nodes, you could, at least in theory, handle a throughput which could be a multiple of a single nodes capability. An approach like this, though, usually requires changes to applications so that they are aware of details of cluster topology and the current node status to detect the write leader.

Read-only routing in PGD Proxy

From PGD 5.5.0, PGD Proxy addresses this requirement to utilize read capacity while minimizing application exposure to the cluster status. It does this by offering a new `read_listen_port` on proxies that complement the existing listen port. Proxies can be configured with either or both of these ports.

When a proxy is configured with a `read_listen_port`, connections to that particular port are routed to available data nodes that aren't the current write leader. If an application only queries and reads from the database, using a `read_listen_port` ensures that your queries aren't answered by the write leader.

Because PGD Proxy is a TCP Layer 4 proxy, it doesn't interfere with traffic passing through it. That means that it can't detect attempts to write passing through the `read_listen_port` connections. As it can't distinguish between a SELECT or an INSERT, it's possible to write through a read-only port.

The active-active nature of PGD means that any write operation will be performed and replicated, and conflict resolution may or may not have to take place. It's up to the application to avoid this and make sure that it uses only `read_listen_ports` for read-only traffic.

Where available, the problem can be mitigated on the client side by passing `default_transaction_read_only=on` in the connection string or equivalent for the driver in use.

Valid read-only nodes

Only data nodes that aren't the write leader are valid as read-only nodes. For reference, the following node types aren't eligible to be a read-only node:

- Witness nodes, because they don't contain data
- Logical standbys, because they're standbys and prioritize replicating
- Subscriber-only nodes

Creating a proxy configuration

SQL proxy creation functions in PGD take an optional `proxy_mode` parameter. You can set this parameter to one of the following values:

- `default` — This is the default value. It creates a proxy that can handle traffic that follows the write leader on port 6432.
- `read-only` — This option creates a read-only proxy that routes traffic to nodes that aren't the write leader. It handles this read-only traffic only on port 6433.
- `any` — This option creates create a proxy that can handle both read-only and write-leader-following traffic on separate ports: 6432 for write-leader-following traffic and 6433 for read-only traffic.

PGD CLI proxy creation passes the `proxy_mode` value using the `--proxy-mode` flag.

Creating a read-only proxy

Using SQL

To create a new read-only proxy, use the `bdr.create_proxy` function:

```
SELECT bdr.create_proxy('proxy-ro1','group-a','read-only');
```

This command creates a read-only proxy named `proxy-ro1` in group `group-a`. By default, it listens on port 6433 for read-only traffic.

Using PGD CLI

To create a new read-only proxy, use the `pgd create-proxy` command with the optional `--proxy-mode` flag set to `read-only`:

```
pgd create-proxy --proxy-name proxy-ro1 --node-group group-a --proxy-mode read-
only
```

Configuring running proxies

> **Note**
>
> After changing a proxy's configuration, restart the proxy to make the changes take effect.

You activate read-only routing on a proxy by setting the `read_listen_port` option to a port number. This port number is the port on which the proxy will listen for read-only traffic. If the proxy already has a `listen_port` set, then the proxy will listen on both ports, routing read/write and read-only traffic respectively on each port. This is equivalent to creating a proxy with `proxy-mode` set to `any`.

If you set a `read_listen_port` on a proxy and then set the `listen_port` to 0, the proxy listens only on the `read_listen_port` and routes only read-only traffic. This is equivalent to creating a proxy with `proxy-mode` set to `read-only`. The configuration elements related to the read/write port are cleared (set to null).

If you set a `listen_port` on a proxy and then set the `read_listen_port` to 0, the proxy listens only on the `listen_port` and routes only read/write traffic. This is equivalent to creating a proxy with `proxy-mode` set to `default`. The configuration elements related to the read-only port are cleared (set to null).

Configuring using SQL

To configure a read-only proxy port on a proxy, use the `bdr.alter_proxy_options` function:

```
SELECT bdr.alter_proxy_options('proxy-
a1','read_listen_port','6433');
```

This command configures a read-only proxy port on port 6433 in the proxy-a1 configuration.

To remove the read-only proxy, set the port to 0:

```
SELECT bdr.alter_proxy_options('proxy-
a1','read_listen_port','0');
```

Configuring using PGD CLI

To configure a read-only proxy port on a proxy, use the `pgd alter-proxy` command:

```
pgd set-proxy-options --proxy-name proxy-a1 --option
read_listen_port=6433
```

This command configures a read-only proxy port on port 6433 in the proxy-a1 configuration.

To remove the read-only proxy, set the port to 0:

```
pgd set-proxy-options --proxy-name proxy-a1 --option
read_listen_port=0
```

14.7 Proxies, Raft, and Raft subgroups

PGD manages its metadata using a Raft model where a top-level group spans all the data nodes in the PGD installation. A Raft leader is elected by the top-level group and propagates the state of the top-level group to all the other nodes in the group.

> **What is Raft?**
>
> Raft is an industry-accepted algorithm for making decisions though achieving *consensus* from a group of separate nodes in a distributed system.

For certain operations in the top-level group, it's essential that a Raft leader must be both established and connected. Examples of these operations include adding and removing nodes and allocating ranges for galloc sequences.

It also means that an absolute majority of nodes in the top-level group (one half of them plus one) must be able to reach each other. So, in a top-level group with five nodes, at least three of the nodes must be reachable by each other to establish a Raft leader.

Proxy routing

One function that also uses Raft is proxy routing. Proxy routing requires that the proxies can coordinate writing to a data node within their group of nodes. This data node is the write leader. If the write leader goes offline, the proxies need to be able to switch to a new write leader, selected by the data nodes, to maintain continuity for connected applications.

You can configure proxy routing on a per-node group basis in PGD 5, but the recommended configurations are *global* and *local* routing.

Global routing

Global routing uses the top-level group to manage the proxy routing. All writable data nodes (not witness or subscribe-only nodes) in the group are eligible to become write leader for all proxies. Connections to proxies within the top-level group will be routed to data nodes within the top-level group.

With global routing, there's only one write leader for the entire top-level group.

Local routing

Local routing uses subgroups, often mapped to locations, to manage the proxy routing within the subgroup. Local routing is often used for geographical separation of writes. It's important for them to continue routing even when the top-level consensus is lost.

That's because PGD allows queries and asynchronous data manipulation (DMLs) to work even when the top-level consensus is lost. But using the top-level consensus, as is the case with global routing, means that new write leaders can't be elected when that consensus is lost. Local groups can't rely on the top-level consensus without adding an independent consensus mechanism and its added complexity.

PGD 5 introduced subgroup Raft support to elegantly address this issue. Subgroup Raft support allows the subgroups in a PGD top-level group to elect the leaders they need independently. They do this by forming devolved Raft groups that can elect write leaders independent of other subgroups or the top-level Raft consensus. Connections to proxies in the subgroup then route to data nodes within the subgroup.

With local routing, there's a write leader for each subgroup.

More information

- Raft subgroups and TPA shows how Raft subgroups can be enabled in PGD when deploying with Trusted Postgres Architect.
- Raft subgroups and PGD CLI shows how the PGD CLI reports on the presence and status of Raft subgroups.
- Migrating to Raft subgroups is a guide to migrating existing installations and enabling Raft subgroups without TPA.
- Raft elections in depth looks in detail at how the write leader is elected using Raft.

14.7.1 Creating Raft subgroups using TPA

The `TPAexec configure` command enables Raft subgroups if the `--enable_proxy_routing local` option is set. TPA uses the term *locations* to reflect the common use case of subgroups that map to physical/regional domains. When the configuration is generated, the location name given is stored under the generated group name, which is based on the location name.

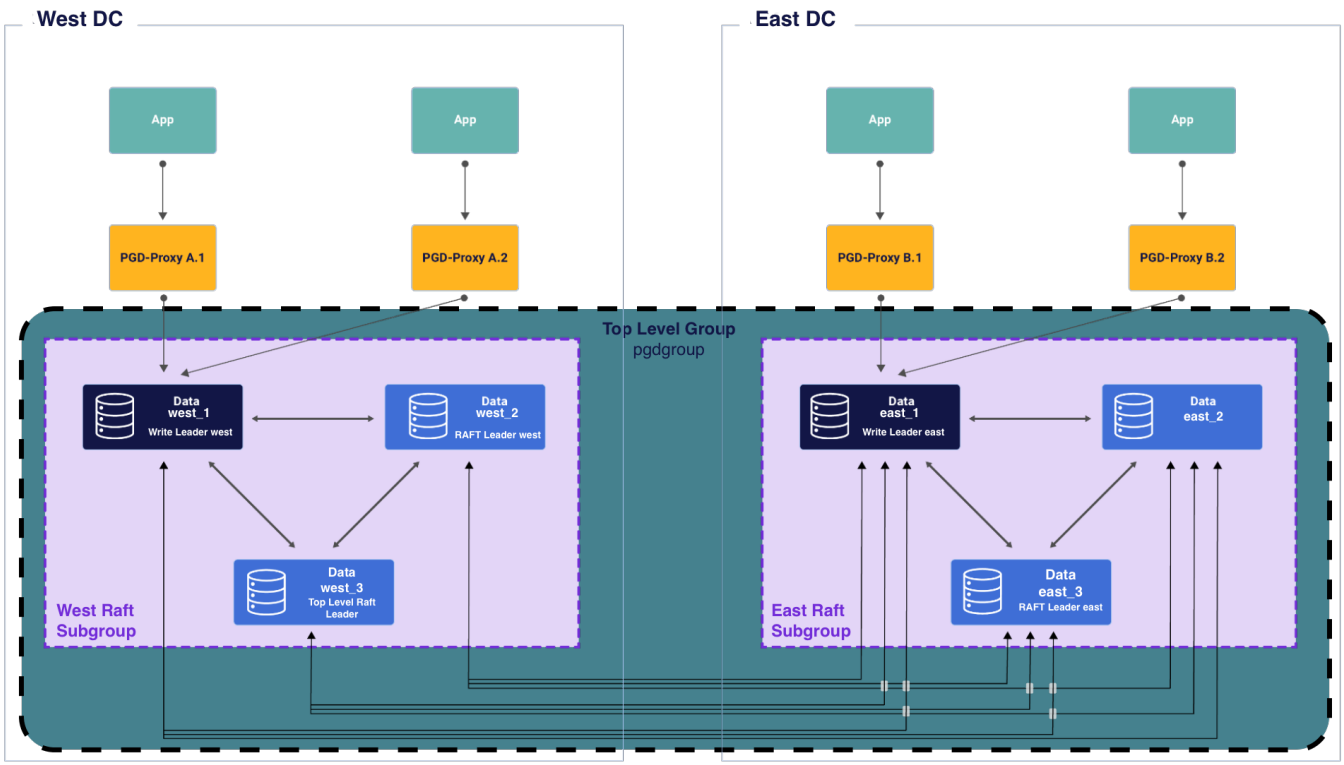
Creating Raft subgroups using TPA

This example creates a two-location cluster with three data nodes in each location. The nodes in each location are part of a PGD Raft subgroup for the location.

The top-level group's name is `pgdgroup`.

The top-level group has two locations: `us_east` and `us_west`. These locations are mapped to two subgroups: `us_east_subgroup` and `us_west_subgroup`.

Each location has four nodes: three data nodes and a barman backup node. The three data nodes also cohost PGD Proxy. The configuration can be visualized like this:



The barman nodes don't participate in the subgroup and, by extension, the Raft group. They're therefore not shown. This diagram is a snapshot of a potential state of the cluster with the West Raft group having selected west_1 as write leader and west_2 as its own Raft leader. On the East, east_1 is write leader while east_3 is Raft leader. The entire cluster is contained within the top-level Raft group. There, west_3 is currently Raft leader.

To create this configuration, you run:

```
tpaexec configure pgdgroup --architecture PGD-Always-ON --location-names us_east us_west --data-nodes-per-
location 3 --epas 16 --no-redwood --enable_proxy_routing local --hostnames-from hostnames.txt
```

Where `hostnames.txt` contains:

```
 east1
east2
east3
eastbarman
west1
west2
west3
westbarman
```

The configuration file

The generated `config.yml` file has a `bdr_node_groups` section that contains the top-level group `pgdgroup` and the two subgroups `us_east_subgroup` and `us_west_subgroup`. Each of those subgroups has a location set (`us_east` and `us_west`) and two other options that are set to true:

- `enable_raft`, which activates the subgroup Raft in the subgroup
- `enable_proxy_routing`, which enables the pgd_proxy routers to route traffic to the subgroup's write leader

Here's an example generated by the sample tpaexec command:

```
cluster_vars:
  apt_repository_list: []
  bdr_database: bdrdb
  bdr_node_group: pgdgroup
  bdr_node_groups:
  - name: pgdgroup
  - name: us_east_subgroup
    options:
      enable_proxy_routing: true
      enable_raft: true
      location: us_east
    parent_group_name: pgdgroup
  - name: us_west_subgroup
    options:
      enable_proxy_routing: true
      enable_raft: true
      location: us_west
    parent_group_name: pgdgroup
  bdr_version: '5'
```

Every node instance has an entry in the instances list. In that entry, `bdr_child_group` appears in the variables section, set to the subgroup the node belongs to. Here's an example generated by the sample tpaexec command:

```
instances:
- Name: east1
  backup: eastbarman
  location: us_east
  node: 1
  role:
  - bdr
  - pgd-proxy
  vars:
    bdr_child_group: us_east_subgroup
    bdr_node_options:
      route_priority: 100
- Name: east2
  location: us_east
  node: 2
  role:
  - bdr
  - pgd-proxy
  vars:
    bdr_child_group: us_east_subgroup
    bdr_node_options:
      route_priority: 100
- Name: east3
  location: us_east
  node: 3
  role:
  - bdr
  - pgd-proxy
  vars:
    bdr_child_group: us_east_subgroup
    bdr_node_options:
      route_priority: 100
- Name: eastbarman
  location: us_east
  node: 4
  role:
  - barman
```

The one node in this location that doesn't have a `bdr_child_group` setting is the barman node because it doesn't participate in the Raft decision-making process.

14.7.2 Working with Raft subgroups and PGD CLI

You can view the status of your nodes and subgroups with the pgd CLI command. The examples here assume a cluster as configured in Creating Raft subgroups with TPA.

Viewing nodes with pgd

The pgd command is `show-nodes`.

```
$pgd show-nodes
```

output

```
Node     Node ID    Node Group Type Current State Target State Status Seq ID
----     -------    ---------- ---- ------------- ------------ ------ ------
east1    916860695  us_east    data ACTIVE        ACTIVE       Up     1
east2    2241077170 us_east    data ACTIVE        ACTIVE       Up     2
east3    1093023575 us_east    data ACTIVE        ACTIVE       Up     3
west1    1668243030 us_west    data ACTIVE        ACTIVE       Up     4
west2    2311995928 us_west    data ACTIVE        ACTIVE       Up     5
west3    4162758468 us_west    data ACTIVE        ACTIVE       Up     6
```

Viewing groups (and subgroups) with pgd

To show the groups in a PGD deployment, along with their names and attributes, use the PGD CLI command `show-groups.`

```
$pgd show-groups
Group    Group ID   Type   Parent Group Location Raft Routing Write Leader
-----    --------   ----   ------------ -------- ---- ------- ------------
pgdgroup 1360502012 global                       true false
us_east  1806884964 data   pgdgroup     us_east  true true    east2
us_west  3098768667 data   pgdgroup     us_west  true true    west1
```

14.7.3 Migrating to Raft subgroups

You can introduce Raft subgroups in a running PGD installation.

Migrating to Raft subgroups (using SQL only)

To enable Raft subgroups to an existing cluster, these configuration steps are needed:

- Identify the top-level group for all nodes in the PGD cluster. An existing cluster already has a top-level group that all nodes belong to.
- Create a subgroup for each location. Use `bdr.create_node_group` with a `parent_group_name` argument that gives the top-level group as its value.
- Add each node at each location to their location's subgroup using `bdr.switch_node_group()`.
- Alter each of the location's subgroups to enable Raft for the group. Use `bdr.alter_node_group_option()`, setting the `enable_raft` option to `true`.

Enabling subgroup Raft node group (using SQL only)

```
SELECT bdr.alter_node_group_option('$group_name', 'enable_raft', 'true');
```

14.7.4 Raft elections in depth

The selection of a write leader in PGD relies on PGD's Raft mechanism. The Raft mechanism is completely internal to PGD's BDR Postgres extension and operates transparently. The nodes within a group begin by establishing a Raft leader within the nodes of the group.

Node interaction

With the Raft leader established, the leader then queries the catalog to see if a write leader for proxy routing was designated.

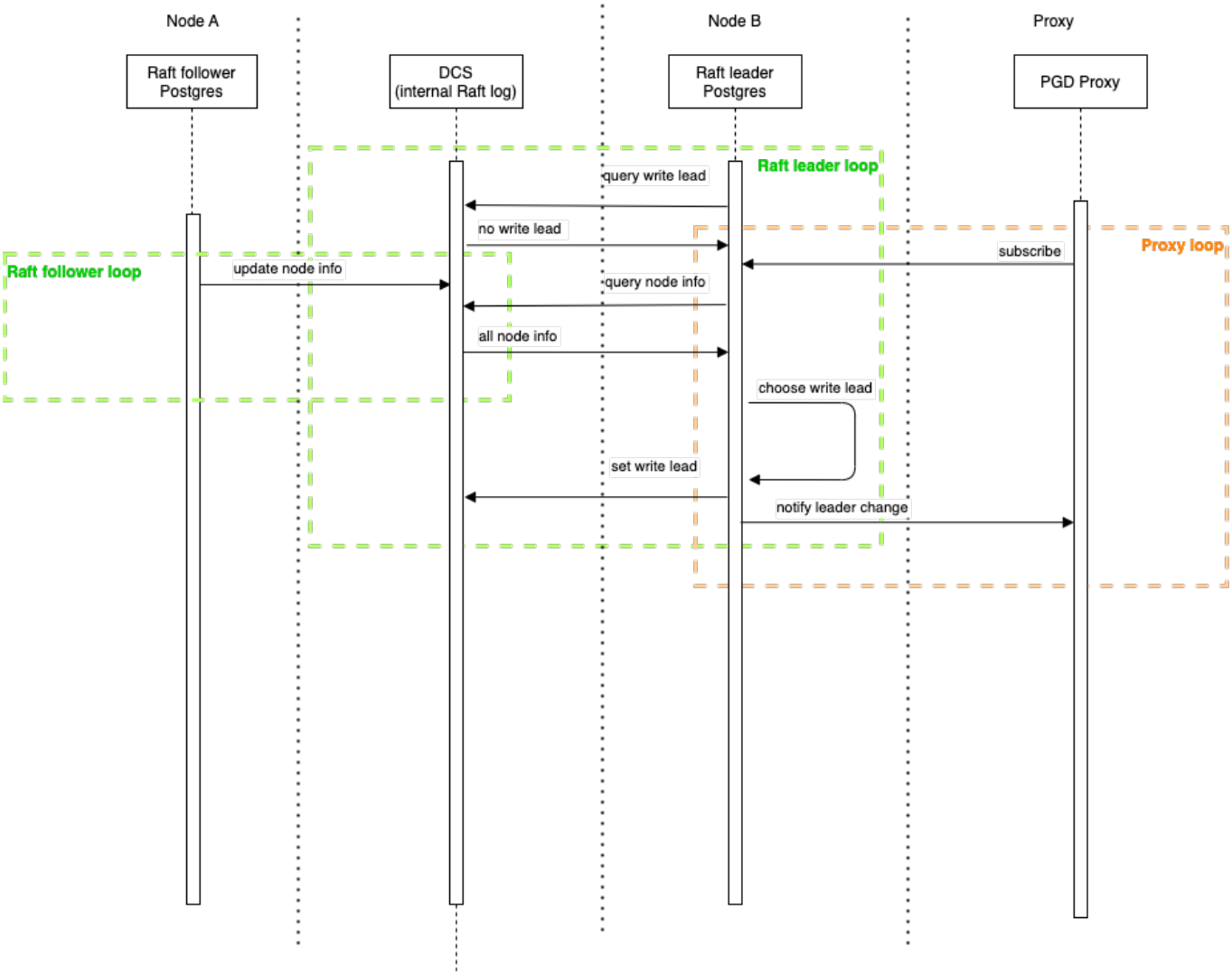
If no write leader is designated, the Raft leader takes steps to designate a new write leader. The process starts by querying all the nodes in the group to establish their state. The resulting list of nodes is then filtered for ineligible nodes (for example, witness nodes) and prioritized. The first/top entry on the list is then set as the new write leader in the Raft log.

Proxy interaction

All proxies initially connect any data node in their group. This behavior allows them to query the catalog for the current write leader and begin routing connections to that node.

They connect to the Raft leader and listen for changes to the catalog entry for write leader. When notified of a change in write leader, they reconfigure routing and send connections to the new write leader.

Both the node and proxy interaction are shown on the following sequence diagram. Two nodes and one proxy are involved, coordinating which node will be write leader and the proxy waiting to learn which node is write leader.



15 Backup and recovery

PGD 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 PGD 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

You can use pg_dump, sometimes referred to as *logical backup*, normally with PGD.

pg_dump dumps both local and global sequences as if they were local sequences. This behavior is intentional, to allow a PGD 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 effectively resets 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 a SQL script like this:

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

If you run pg_dump using `bdr.crdt_raw_value = on`, then you can reload the dump only with `bdr.crdt_raw_value = on`.

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

Physical backup

You can take physical backups of a node in an EDB Postgres Distributed cluster using standard PostgreSQL software, such as Barman.

You can perform a physical backup of a PGD node using the same procedure that applies to any PostgreSQL node. A PGD node is just a PostgreSQL node running the BDR extension.

Consider these specific points when applying PostgreSQL backup techniques to PGD:

- PGD operates at the level of a single database, while a physical backup includes all the databases in the instance. Plan your databases to allow them to be easily backed up and restored.
- Backups make a copy of just one node. In the simplest case, every node has a copy of all data, so you need to back up only one node to capture all data. However, the goal of PGD isn't met if the site containing that single copy goes down, so the minimum is at least one node backup per site (with many copies, and so on).
- However, each node might have unreplicated local data, or the definition of replication sets might be complex so that all nodes don't subscribe to all replication sets. In these cases, backup planning must also include plans for how to back up any unreplicated local data and a backup of at least one node that subscribes to each replication set.

Eventual consistency

The nodes in an EDB Postgres Distributed cluster are *eventually consistent* but not *entirely consistent*. A physical backup of a given node provides point-in-time recovery capabilities limited to the states actually assumed by that node.

The following example shows how two nodes in the same EDB Postgres Distributed cluster might not (and usually don't) go through the same sequence of states.

Consider a cluster with two nodes, `N1` and `N2`, that's 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` goes through the following states:

```
(N1)   S  -->  S + W1  -->  S + W1 + W2
```

Node `N2` goes through the following states:

```
(N2)   S  -->  S + W2  -->  S + W1 + W2
```

That is, node `N1` *never* assumes state `S + W2`, and node `N2` likewise never assumes state `S + W1`. However, both nodes end up in the same state `S + W1 + W2`. Considering this situation might affect how you decide on your backup strategy.

Point-in-time recovery (PITR)

The previous example showed that the changes are also inconsistent in time. `W1` and `W2` both occur at time `T1`, but the change `W1` isn't 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 scanning through changes until one particular point in time (PIT) is reached. With this scheme, you can restore one node to a single PIT from its viewpoint, for example, `T1`. However, that state doesn't include other data from other nodes that 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.

With PostgreSQL PITR, you can use the standard syntax:

```
recovery_target_time = T1
```

PGD allows for changes from multiple masters, all recorded in the WAL log for one node, separately identified using replication origin identifiers.

PGD 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, you can view the WAL stream as containing multiple streams all mixed up into one larger stream. There's still just one PIT, but that's reached as different points for each origin separately.

The WAL stream is read until requested origins have found their PIT. All changes are applied up until that point, except that any transaction records aren't marked as committed for an origin after the PIT on that origin is reached.

You end up with one LSN "stopping point" in WAL, but you also have one single timestamp applied consistently, just as you do with single-origin PITR.

Once you reach the defined PIT, a later one might also be set to allow the recovery to continue, as needed.

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

In the specific example shown, `N1` is restored to `T1`. It also includes changes from other nodes that were committed by `T1`, even though they weren't applied on `N1` until later.

To request multi-origin PITR, use the standard syntax in the `postgresql.conf` file:

```
recovery_target_time = T1
```

You need to specify the list of replication origins that are restored to `T1` in one of two ways. You can use a separate `multi_recovery.conf` file by way of a new parameter, `recovery_target_origins`:

```
recovery_target_origins = '*'
```

Or you can specify the origin subset as a list in `recovery_target_origins`:

```
recovery_target_origins = '1,3'
```

The local WAL activity recovery to the specified `recovery_target_time` is always performed implicitly. For origins that aren't specified in `recovery_target_origins`, recovery can 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 behavior that's designed around the assumption of changes arriving from a single master in COMMIT order.

> **Note**
>
> This feature is available only with EDB Postgres Extended. Barman doesn't create a `multi_recovery.conf` file.

Restore

While you can take a physical backup with the same procedure as a standard PostgreSQL node, it's slightly more complex to restore the physical backup of a PGD 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 PGD nodes in a cluster, for instance in the event of a data center failure.

You might 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, you can restore PGD capabilities based on a physical backup of a single PGD node, optionally plus WAL archives:

- If you still have some PGD nodes live and running, fence off the host you restored the PGD node to, so it can't connect to any surviving PGD nodes. This practice ensures that the new node doesn't confuse the existing cluster.
- Restore a single PostgreSQL node from a physical backup of one of the PGD nodes.
- If you have WAL archives associated with the backup, create a suitable `postgresql.conf`, and start PostgreSQL in recovery to replay up to the latest state. You can specify an 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 PGD metadata that was included in the physical backup.
- Fully stop and restart the PostgreSQL instance.
- Add further PGD nodes with the standard procedure based on the `bdr.join_node_group()` function call.

Cleanup of PGD metadata

To clean up leftover PGD metadata:

1. Drop the PGD node using `bdr.drop_node`.
2. Fully stop and restart PostgreSQL (important!).

Cleanup of replication origins

You must explicitly remove replication origins with a separate step because they're recorded persistently in a system catalog. They're therefore included in the backup and in the restored instance. They aren't removed automatically when dropping the BDR extension because they aren't explicitly recorded as its dependencies.

To track progress of incoming replication in a crash-safe way, PGD creates one replication origin for each remote master node. Therefore, for each node in the previous cluster run this once:

```
SELECT pg_replication_origin_drop('bdr_dbname_grpname_nodename');
```

You can list replication origins as follows:

```
SELECT * FROM pg_replication_origin;
```

Those created by PGD are easily recognized by their name.

Cleanup of replication slots

If a physical backup was created with `pg_basebackup`, replication slots are omitted from the backup.

Some other backup methods might preserve replications slots, likely in outdated or invalid states. Once you restore the backup, use these commands to drop all replication slots:

```
SELECT pg_drop_replication_slot(slot_name)
FROM pg_replication_slots;
```

If you have a reason to preserve some slots, you can add a `WHERE slot_name LIKE 'bdr%'` clause, but this is rarely useful.

> **Warning**
>
> Never use these commands to drop replication slots on a live PGD node

16 Security and roles

EDB Postgres Distributed allows a PGD cluster to be administered without giving access to the stored data by design. It achieves this through the use of roles and controlled access to system objects.

- Roles introduces the roles that PGD predefines for controlling access to PGD functionality.
- Role management discusses how roles are managed on multi-database nodes and new nodes.
- PGD predefined roles details the specific privileges of the PGD roles.
- Roles and replication explains how PGD replication interacts with roles and privileges.
- Access control explains how tables, functions, catalog objects and triggers interact with PGD roles and Postgres attributes.

16.1 Roles

Configuring and managing PGD doesn't require superuser access and we recommend that you don't use superuser access. Instead, the privileges required to administer PGD are split across the following predefined roles.

Role	Description
bdr_superuser	The highest-privileged role, having access to all PGD 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 PGD.
bdr_monitor	Includes the privileges of bdr_read_all_stats, with some extra privileges for monitoring.
bdr_application	The minimal privileges required by applications running PGD.
bdr_read_all_conflicts	Can view all conflicts in `bdr.conflict_history`.

These roles are named to be analogous to PostgreSQL's `pg_` predefined roles.

The PGD `bdr_` roles are created when the BDR extension is installed. See PGD predefined roles for more details of the privileges each role has.

Managing PGD doesn't require that administrators have access to user data.

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

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

> **The BDR extension and superuser access**
>
> The one exception to the rule of not needing superuser access is in the management of PGD's underlying BDR extension. 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.

16.2 Role management

Users are global objects in a PostgreSQL instance. A `CREATE ROLE` command or its alias `CREATE USER` is replicated automatically if it's executed in a PGD replicated database. If a role or user is created in a non-PGD, unreplicated database, the role exists only for that PostgreSQL instance. `GRANT ROLE` and `DROP ROLE` work the same way, replicating only if applied to a PGD-replicated database.

> **Note**
>
> Remember that a user in Postgres terms is simply a role with login privileges.

Role rule - No un-replicated roles

If you do create a role or user in a non-PGD, unreplicated database, it's especially important that you do not make an object in the PGD-replicated database rely on that role. It will break the replication process, as PGD cannot replicate a role that is not in the PGD-replicated database.

You can disable this automatic replication behavior by turning off the `bdr.role_replication` setting, but we don't recommend that.

Roles for new nodes

New PGD nodes that are added using `bdr_init_physical` will automatically replicate the roles from other nodes of the PGD cluster.

If a PGD node is joined to a PGD group manually, without using `bdr_init_physical`, existing roles aren't copied to the newly joined node. This is intentional behavior to ensure that access isn't accidentally granted.

Roles, by default, have access to all databases. Automatically copying users to a newly created node would therefore open the possibility of unintentionally exposing a database that exists on the node. If the PGD cluster had a user Alice, and Alice was automatically created on a newly joined node, by directly connecting to that node, Alice would have access to all the databases on that node. PGD therefore doesn't copy over existing roles. In this situation, we recommend that you copy over roles manually.

PostgreSQL allows you to dump all roles with the command:

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

You can then edit the file `roles.sql` to remove unwanted users before running the file's commands on the newly created node.

When joining a new node, the “No unreplicated roles” rule also applies. If an object in the database to be replicated relies on an unreplicated role in an (unreplicated) local database, then the join fails when it attempts to replicate that object.

Connections and roles

When allocating a new PGD 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.

PGD 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 these 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's 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.

Also, if any non-default extensions (excluding the BDR extension) are present on the source node, and any of these can be installed only by a superuser, a superuser must create these extensions manually on the join target node. Otherwise the join process will fail.

In PostgreSQL 13 and later, you can identify the extensions requiring superuser permission and that must be manually installed. On the source node, execute:

```
    SELECT name, (trusted IS FALSE AND superuser) AS
superuser_only
     FROM
pg_available_extension_versions
    WHERE installed AND name != 'bdr';
```

Once all nodes are joined, to continue to allow DML and DDL replication, you can further reduce the permissions to the following:

- The user has the `REPLICATION` attribute.
- It inherits the bdr_superuser role.

16.3 PGD predefined roles

PGD predefined roles are created when the BDR extension is installed. After BDR extension is dropped from a database, the roles continue to exist. You need to drop them manually if dropping is required.

bdr_superuser

This is a role for an admin user that can manage anything PGD related. It allows you to separate management of the database and table access. Using it allows you to have a user that can manage the PGD cluster without giving them PostgreSQL superuser privileges.

Privileges

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

bdr_read_all_stats

This role provides read access to most of the tables, views, and functions that users or applications may need to observe the statistics and state of the PGD cluster.

Privileges

`SELECT` privilege on:

- `bdr.autopartition_partitions`
- `bdr.autopartition_rules`
- `bdr.ddl_epoch`
- `bdr.ddl_replication`
- `bdr.global_consensus_journal_details`
- `bdr.global_lock`
- `bdr.global_locks`
- `bdr.group_camo_details`
- `bdr.local_consensus_state`
- `bdr.local_node_summary`
- `bdr.node`
- `bdr.node_catchup_info`
- `bdr.node_catchup_info_details`
- `bdr.node_conflict_resolvers`
- `bdr.node_group`
- `bdr.node_local_info`
- `bdr.node_peer_progress`
- `bdr.node_replication_rates`
- `bdr.node_slots`
- `bdr.node_summary`
- `bdr.replication_sets`
- `bdr.replication_status`
- `bdr.sequences`
- `bdr.stat_activity`
- `bdr.stat_relation`
- `bdr.stat_subscription` *deprecated*
- `bdr.state_journal_details`
- `bdr.subscription`
- `bdr.subscription_summary`
- `bdr.tables`
- `bdr.taskmgr_local_work_queue`
- `bdr.taskmgr_work_queue`

- `bdr.worker_errors` *deprecated*
- `bdr.workers`
- `bdr.writers`
- `bdr.xid_peer_progress`

EXECUTE privilege on:

- `bdr.bdr_edition` *deprecated*
- `bdr.bdr_version`
- `bdr.bdr_version_num`
- `bdr.decode_message_payload`
- `bdr.get_consensus_status`
- `bdr.get_decoding_worker_stat`
- `bdr.get_global_locks`
- `bdr.get_min_required_replication_slots`
- `bdr.get_min_required_worker_processes`
- `bdr.get_raft_status`
- `bdr.get_relation_stats`
- `bdr.get_slot_flush_timestamp`
- `bdr.get_sub_progress_timestamp`
- `bdr.get_subscription_stats`
- `bdr.lag_control`
- `bdr.lag_history`
- `bdr.node_catchup_state_name`
- `bdr.node_kind_name`
- `bdr.peer_state_name`
- `bdr.pglogical_proto_version_ranges`
- `bdr.show_subscription_status`
- `bdr.show_workers`
- `bdr.show_writers`
- `bdr.stat_get_activity`
- `bdr.wal_sender_stats`
- `bdr.worker_role_id_name`

bdr_monitor

This role provides read access to any tables, views, and functions that users or applications may need to monitor the PGD cluster. It includes all the privileges of the `bdr_read_all_stats` role.

Privileges

All privileges from `bdr_read_all_stats` plus the following additional privileges:

`SELECT` privilege on:

- `bdr.group_raft_details`
- `bdr.group_replslots_details`
- `bdr.group_subscription_summary`
- `bdr.group_versions_details`
- `bdr.raft_instances`

`EXECUTE` privilege on:

- `bdr.get_raft_instance_by_nodegroup`
- `bdr.monitor_camo_on_all_nodes`
- `bdr.monitor_group_raft`
- `bdr.monitor_group_versions`
- `bdr.monitor_local_replslots`
- `bdr.monitor_raft_details_on_all_nodes`

- `bdr.monitor_replslots_details_on_all_nodes`
- `bdr.monitor_subscription_details_on_all_nodes`
- `bdr.monitor_version_details_on_all_nodes`
- `bdr.node_group_member_info`

bdr_application

This role is designed for applications that require access to PGD features, objects, and functions such as sequences, CRDT datatypes, CAMO status functions, or trigger management functions.

Privileges

`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 require additional privileges 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

PGD logs conflicts into the `bdr.conflict_history` table. Conflicts are visible only to table owners, so no extra privileges are required for the owners to read the conflict history.

If, though, 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.

Privileges

An explicit policy is set on `bdr.conflict_history` that allows this role to read the `bdr.conflict_history` table.

16.4 Roles and replication

DDL and DML replication and users

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.

By default, PGD replicates new tables with the same owner across nodes.

Differing table ownership

We recommend for the same user to own the table on each node. That's the default behavior, but you can override it. If you do, there are some things to take into account.

Consider a situation where 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 nodes can have different use cases, we do allow this scenario. But we also warn against it. If tables have different owners on different nodes, we recommend that a security administrator help to plan and audit this configuration.

Replication and row-level security

On tables with row-level security policies enabled, changes are replicated without reenforcing policies on apply. This behavior 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 that the row security policies on all nodes be identical or at least compatible, but we don't enforce this.

bdr_superuser role and replication

The user bdr_superuser controls replication for PGD and can add or remove any table from any replication set. bdr_superuser doesn't need any privileges over individual tables, nor do we recommend it. If you need to restrict access to replication set functions, you can implement restricted versions of these functions as `SECURITY DEFINER` functions and grant them to the appropriate users.

Privilege restrictions

PGD 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.

This means that the REFERENCES privilege alone isn't sufficient to allow creating a foreign key with PGD. 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's typically too expensive to use successfully.

16.5 Access control

Catalog tables

System catalog and information schema tables are always excluded from replication by PGD.

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

PGD functions and operators

All PGD 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 PGD 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 PGD 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 PGD and therefore of the PGD 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's joining.

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 PGD, 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."

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

- superuser: Can create trigggers.
- bdr_superuser: Can create triggers.
- 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 use a function that's owned by the same owner as the table and they satisfy standard PostgreSQL rules. Specifically, they must have EXECUTE privilege on the function.

  If both table and function have the same owner, and the owner decides to give a user both TRIGGER privilege on the table and EXECUTE privilege on thae function. It's assumed that it's okay for that user to create a trigger on that table using this function.
- Users who have TRIGGER privilege on the table: Can also create triggers using functions that are defined with the SECURITY DEFINER clause if they have EXECUTE privilege on them.

  The SECURITY DEFINER clause makes the function always execute as the owner of the function both in standard PostgreSQL and PGD.

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.

When PGD replication applies changes it uses the system-level default search_path only. Replica triggers, stream triggers, and index expression functions that assume other search_path settings will then fail when they execute on apply. To ensure this doesn't occur, resolve object references clearly using either the default search_path only, or set the search path for a function using `ALTER FUNCTION ... SET search_path = ...` for the functions affected. When using the default search_path, always use fully qualified references to objects, for example, schema.objectname.

17 Monitoring

Monitoring replication setups is important to ensure that your system:

- Performs optimally
- Doesn't run out of disk space
- Doesn't encounter other faults that might halt operations

It's important to have automated monitoring in place to ensure that the administrator is alerted and can take proactive action when issues occur. For example, the administrator can be alerted if replication slots start falling badly behind.

EDB provides Postgres Enterprise Manager (PEM), which supports PGD starting with version 8.1. See Monitoring EDB Postgres Distributed for more information.

Alternatively, tools or users can make their own calls into information views and functions provided by the BDR extension. See Monitoring through SQL for details.

EDB Postgres Distributed also integrates with OpenTelemetry, allowing you to use an existing reporting setup to follow the state of the EDB Postgres Distributed cluster. See OpenTelemetry integration for details.

17.1 OpenTelemetry integration

You can configure EDB Postgres Distributed to report monitoring information as well as traces to theOpenTelemetry collector.

EDB Postgres Distributed OTEL collector fills several resource attributes. These are attached to all metrics and traces:

- The `service.name` is configurable with the `bdr.otel_service_name` configuration setting.
- The `service.namespace` is always set to `edb_postgres_distributed`.
- The `service.instance.id` is always set to the system identifier of the Postgres instance.
- The `service.version` is set to the current version of the BDR extension loaded in the Postgresql instance.

OTEL and OLTP compatibility

For OTEL connections, the integration supports OLTP/HTTP version 1.0.0 only, over HTTP or HTTPS. It doesn't support OLTP/gRPC.

Metrics collection

Setting the configuration option `bdr.metrics_otel_http_url` to a non-empty URL enables the metric collection.

Different kinds of metrics are collected, as shown in the tables that follow.

Generic metrics

Metric name	Type	Labels	Description
pg_backends_by_state	gauge	conn_state - idle, active, idle in transaction, fastpath functioncall, idle in transaction (aborted), disabled, undefined	Number of backends in a given state
pg_oldest_xact_start	gauge		Oldest transaction start time
pg_oldest_activity_start	gauge		Oldest query start time
pg_waiting_backends	gauge	wait_type - LWLock, Lock, BufferPin, Activity, Client, Extension, IPC, Timeout, IO, ??? (for unknown)	Number of currently waiting backends by wait type
pg_start_time	gauge		Timestamp at which the server has started
pg_reload_time	gauge		Timestamp at which the server has last reloaded configuration

Replication metrics

Metric name	Type	Labels	Description
bdr_slot_sent_lag	gauge	slot_name - name of a slot	Current sent lag in bytes for each replication slot
bdr_slot_write_lag	gauge	slot_name - name of a slot	Current write lag in bytes for each replication slot
bdr_slot_flush_lag	gauge	slot_name - name of a slot	Current flush lag in bytes for each replication slot
bdr_slot_apply_lag	gauge	slot_name - name of a slot	Current apply lag in bytes for each replication slot
bdr_subscription_receive_lsn	gauge	sub_name - name of subscription	Current received LSN for each subscription
bdr_subscription_flush_lsn	gauge	sub_name - name of subscription	Current flushed LSN for each subscription
bdr_subscription_apply_lsn	gauge	sub_name - name of subscription	Current applied LSN for each subscription
bdr_subscription_receiver	gauge	sub_name - name of subscription	Whether subscription receiver is currently running (1) or not (0)

Consensus metric

See also Monitoring Raft consensus

Metric name	Type	Labels	Description
bdr_raft_state	gauge	state_str - RAFT_FOLLOWER, RAFT_CANDIDATE, RAFT_LEADER, RAFT_STOPPED	Raft state of the consensus on this node
bdr_raft_protocol_version	gauge		Consensus protocol version used by this node
bdr_raft_leader_node	gauge		Id of a node that this node considers to be current leader
bdr_raft_nodes	gauge		Total number of nodes that participate in consensus (includes learner/non-voting nodes)
bdr_raft_voting_nodes	gauge		Number of actual voting nodes in consensus
bdr_raft_term	gauge		Current raft term this node is on
bdr_raft_commit_index	gauge		Raft commit index committed by this node
bdr_raft_apply_index	gauge		Raft commit index applied by this node

Tracing

Tracing collection to OpenTelemetry requires configuring `bdr.trace_otel_http_url` and enabling tracing using `bdr.trace_enable`.

The tracing is currently limited to only some subsystems, primarily to the cluster management functionality. The following spans can be seen in traces.

Span name	Description
create_node_group	Group creation
alter_node_group_config	Change of group config options
alter_node_config	Change of node config option
join_node_group	Node joining a group
join_send_remote_request	Join source sending the join request on behalf of the joining node
add_camo_pair	Add CAMO pair
alter_camo_pair	Change CAMO pair
remove_camo_pair	Delete CAMO pair
alter_commit_scope	Change commit scope definition (either create new or update existing)
alter_proxy_config	Change config for PGD-Proxy instance (either create new or update existing)
walmsg_global_lock_send	Send global locking WAL message
walmsg_global_lock_recv	Received global locking WAL message
ddl_epoch_apply	Global locking epoch apply (ensure cluster is synchronized enough for new epoch to start)
walmsg_catchup	Catchup during node removal WAL message
raft_send_appendentries	Internal Raft book keeping message
raft_recv_appendentries	Internal Raft book keeping message
raft_request	Raft request execution
raft_query	Raft query execution
msgb_send	Consensus messaging layer message
msgb_recv_receive	Consensus messaging layer message
msgb_recv_deliver	Consensus messaging layer message delivery
preprocess_ddl	DDL command preprocessing

TLS support

The metrics and tracing endpoints can be HTTP or HTTPS. You can configure paths to the CA bundle, client key, and client certificate using `bdr.otel_https_ca_path`, `bdr.otel_https_key_path`, and `bdr.otel_https_cert_path` configuration options.

17.2 Monitoring through SQL

EDB Postgres Distributed provides several monitoring and statistics views that are specific to its distributed nature. The standard Postgres monitoring is also useful for monitoring EDB Postgres Distributed.

Monitoring overview

A PGD group consists of multiple servers, often referred to as nodes. Monitor all of the nodes to ensure the health of the whole group.

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

- `OK` — Often shown as green.
- `WARNING` — Often shown as yellow.
- `CRITICAL` — Often shown as red.
- `UNKNOWN` — For unrecognized situations, often shown as red.

PGD also provides dynamic catalog views that show the instantaneous state of various internal metrics. It also provides metadata catalogs that store the configuration defaults and configuration changes the user requests. 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.

PGD allows you to monitor each of the nodes individually or to monitor the whole group by access to a single node. If you want to monitor each node individually, connect to each node and issue monitoring requests. If you want 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 were granted access to the `bdr.run_on_all_nodes()` function by bdr_superuser, then you can 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. You can turn waiting 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 using `bdr.node_summary` and `bdr.event_summary`.

When called, the helper function `bdr.wait_for_join_completion()` causes a PostgreSQL session to pause until all outstanding node join operations area complete.

This example shows the output of a `SELECT` query from `bdr.node_summary`. It 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` gives information on the catch-up state, which can be relevant to joining nodes or parting nodes.

When a node is parted, some nodes in the cluster might not receive all the data from that parting node. So parting a node creates 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 the manager worker

The manager worker is responsible for many background tasks, including the managing of all the other workers. As such it is important to know what it's doing, especially in cases where it might seem stuck.

Accordingly, the `bdr.stat_worker` view provides per worker statistics for PGD workers, including manager workers. With respect to ensuring manager workers do not get stuck, the current task they are executing would be reported in their `query` field prefixed by "pgd manager:".

The `worker_backend_state` field for manager workers also reports whether the manager is idle or busy.

Monitoring Routing

Routing is a critical part of PGD for ensuring a seemless application experience and conflict avoidance. Routing changes should happen quickly, including the detections of failures. At the same time we want to have as few disruptions as possible. We also want to ensure good load balancing for use-cases where it's supported.

Monitoring all of these is important for noticing issues, debugging issues, as well as informing more optimal configurations. Accoringly, there are two main views for monitoring statistics to do with routing:

- `bdr.stat_routing_state` for monitoring the state of the connection routing with PGD Proxy uses to route the connections.
- `bdr.stat_routing_candidate_state` for information about routing candidate nodes from the point of view of the Raft leader (the view is empty on other nodes).

Monitoring Replication Peers

You use two main views for monitoring of replication activity:

- `bdr.node_slots` for monitoring outgoing replication
- `bdr.subscription_summary` for monitoring incoming replication

You can also obtain most of the information provided by `bdr.node_slots` by querying the standard PostgreSQL replication monitoring views `pg_catalog.pg_stat_replication` and `pg_catalog.pg_replication_slots`.

Each node has one PGD group slot that must never have a connection to it and is very rarely be marked as active. This is normal and doesn't imply something is down or disconnected. See Replication slots in Node Management.

Monitoring outgoing replication

You can use another view 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` refers to the rate in bytes per second. It's 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. This information will be fixed in a future release. As a workaround, we recommend using 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 from the `bdr.node_slots` view.

Administrators can 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 PGD 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
```

Because PGD is a mesh network, to get the full view of lag in the cluster, you must execute this query 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 a human-readable form of the same. It's important to understand that WAL usually contains a lot of writes that aren't replicated but still count in `replay_lag_bytes`, including, for example:

- `VACUUM` activity
- Index changes
- Writes associated with other databases on the same node
- Writes for tables that are not part of a replication set

So the lag in bytes reported here isn't 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` isn't a measure of how long the peer node takes to catch up or how long it takes 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 doesn't advance while logical replication is streaming a transaction. It changes only when a commit is replicated. So the lag tends to "sawtooth," rising as a transaction is streamed and then falling again as the peer node commits it, flushes it, and sends confirmation. The reported LSN positions "stair-step" instead of advancing smoothly, for similar reasons.

When replication is disconnected (`active` = `'f'`), the `active_pid` column is `NULL`, as is `client_addr` and the other fields that make sense only with an active connection. The `state` field is `'disconnected'`. The `_lsn` fields are the same as the `confirmed_flush_lsn`, since that's the last position that the client is known for certain to have replayed to and saved. The `_lag` fields show the elapsed time between the most recent confirmed flush on the client and the current time. The `_lag_size` and `_lag_bytes` fields report the distance between `confirmed_flush_lsn` and the local server's current WAL insert position.

> **Note**
>
> It's normal for `restart_lsn` to be behind the other `lsn` columns. This doesn't 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. It generally reflects the position of the oldest transaction that's 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 it's otherwise harmless. If you're concerned, look for very long-running transactions and forgotten prepared transactions.

Monitoring incoming replication

You can monitor incoming replication (also called subscriptions) at a high level by querying the `bdr.subscription_summary` view. This query shows the list of known subscriptions to other nodes in the EDB Postgres Distributed cluster and the state of the replication worker:

```
 # 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
```

You can further monitor subscriptions by monitoring subscription summary statistics through `bdr.stat_subscription`, and by monitoring the subscription replication receivers and subscription replication writers, using `bdr.stat_receiver` and `bdr.stat_writer`, respectively.

Monitoring WAL senders using LCR

If the decoding worker is enabled, you can monitor information about the current logical change record (LCR) file for each WAL sender using the function `bdr.wal_sender_stats()`. For example:

```
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 | 000000000000000000000014000000000000000000
 2059916 | t            | bdr_postgres_bdrgroup_decoder | 000000000000000000000014000000000000000000
(3 rows)
```

If `is_using_lcr` is `FALSE`, `decoder_slot_name` / `lcr_file_name` is `NULL`. This is the case if the decoding worker isn't enabled or the WAL sender is serving a logical standby.

Also, you can monitor information about the decoding worker using the function `bdr.get_decoding_worker_stat()`. For example:

```
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 PGD replication workers

All PGD 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 PGD system:

- The wait_event column has enhanced information, if the reason for waiting is related to PGD.
- The `query` column is blank in PGD workers, except when a writer process is executing DDL, or for when a manager worker is active (in which case the entry in the `query` column will be prefixed with "`pgd manager:`").

The `bdr.workers` view shows PGD worker-specific details that aren't available from `bdr.stat_activity`.

The view `bdr.event_summary` shows the last error (if any) reported by any worker that has a problem continuing the work. This information is persistent, so it's important to note the time of the error and not just its existence. Most errors are transient, and PGD workers will retry the failed operation.

Monitoring PGD writers

Another system view, `bdr.writers`, monitors writer activities. This view shows only the current status of writer workers. It includes:

- `sub_name` to identify the subscription that the writer belongs to
- `pid` of the writer process
- `streaming_allowed` to know if the writer supports applying 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

PGD honors commit ordering by following the same commit order as happened on the origin. In case of parallel writers, multiple writers might apply different transactions at the same time. The `commit_queue_position` shows the order in which they will commit. Value `0` means that the writer is the first one to commit. Value `-1` means that the commit position isn't yet known, which can happen for a streaming transaction or when the writer isn't currently applying any transaction.

Monitoring commit scopes

Commit scopes are our durability and consistency configuration framework. As such, they affect the performance of transactions, so it is important to get statistics on them. Moreover, because in failure scenarios transactions might appear to be stuck due to the commit scope configuration, we need insight into what commit scope is being used, what it's waiting on, and so on.

Accordingly, these two views show relevant statistics about commit scopes:

- bdr.stat_commit_scope for cumulative statistics for each commit scope.
- bdr.stat_commit_scope_state for information about the current use of commit scopes by backend processes.

Monitoring global locks

The global lock, which is currently used only for DDL replication, is a heavyweight lock that exists across the whole PGD group.

There are currently two types of global locks:

- DDL lock, used for serializing all DDL operations on permanent (not temporary) objects (that is, tables) in the database
- DML relation lock, used for locking out writes to relations during DDL operations that change the relation definition

You can create either or both entry types 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 the relation that's being locked, the PID holding the lock (if local), and whether the lock was globally granted. In case of global advisory locks, `lock_type` column shows `GLOBAL_LOCK_ADVISORY`, and `relation` column shows the advisory keys on which the lock is acquired.

This example shows the output of `bdr.global_locks` while running an `ALTER TABLE` statement with `bdr.ddl_locking = 'all'`:

```
 # 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 Catalogs 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. Monitor the PGD system to identify conflicts and, where possible, make application changes to eliminate the conflicts or make them less frequent.

By default, all conflicts are logged to `bdr.conflict_history`. Since this log contains full details of conflicting data, the rows are protected by row-level security to ensure they're 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 resolve.

For monitoring purposes, use `bdr.conflict_history_summary`, which doesn't contain user data. This example shows a query to count the number of conflicts seen in the current day using an efficient query plan:

```
 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');
```

Apply statistics

PGD 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

For relations only, `bdr.stat_relation` also includes:

- Total time spent processing replication for the relation
- Total lock wait time to acquire lock (if any) for the relation (only)

For subscriptions only, `bdr.stat_subscription` includes:

- Number of COMMITs/DDL replicated for the subscription
- Number of times this subscription has connected upstream

Tracking of these statistics is controlled by the PGD GUCs `bdr.track_subscription_apply` and `bdr.track_relation_apply`, respectively.

The following shows the example output from these:

```
 # 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 three times to the upstream, inserted 10 rows, and performed two DDL commands inside five transactions.

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

PGD also monitors statistics regarding subscription replication receivers and subscription replication writers for each subscription, using `bdr.stat_receiver` and `bdr.stat_writer`, respectively.

Standard PostgreSQL statistics views

Statistics on table and index usage are normally updated 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 haven't been reset, these two views 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 PGD group, the statistics for the same table in the new node show 1M inserts and zero updates. However, from that moment, the upstream and downstream table statistics change by the same amounts because all changes on one side are replicated to the other side.

Since indexes are used to apply changes, the identifying indexes on the downstream side might appear more heavily used with workloads that perform `UPDATE` and `DELETE` 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 PGD versions

PGD allows running different Postgres versions as well as different BDR extension versions across the nodes in the same cluster. This capability is useful for upgrading.

The view `bdr.group_versions_details` uses the function `bdr.run_on_all_nodes()` to retrieve Postgres and BDR extension 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     | 15.2.0           |
5.0.0
 node2     | 15.2.0           |
5.0.0
```

The recommended setup is to try to have all nodes running the same (and latest) versions as soon as possible. We recommend that the cluster doesn't run different versions of the BDR extension for too long.

For monitoring purposes, we recommend the following alert levels:

- status=UNKNOWN, message=This node is not part of any PGD group
- status=OK, message=All nodes are running same PGD versions
- status=WARNING, message=There is at least 1 node that is not accessible
- status=WARNING, message=There are node(s) running different PGD versions when compared to other nodes

The described behavior is implemented in the function `bdr.monitor_group_versions()`, which uses PGD 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 must be working cluster-wide at all times. The impact of running an EDB Postgres Distributed cluster without Raft consensus working might be as follows:

- The replication of PGD data changes might still work correctly.
- Global DDL/DML locks doesn't work.
- Galloc sequences eventually run out of chunks.
- Eager Replication doesn't work.
- Cluster maintenance operations (join node, part node, promote standby) are still allowed, but they might not finish (hanging instead).
- Node statuses might not be correctly synced among the PGD nodes.
- PGD group replication slot doesn't advance LSN and 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 | node_group_name |     state     |
leader_id
------------+-----------+-----------------+---------------+------------
 1148549230 | node1     | top_group       | RAFT_LEADER   |
1148549230
 3367056606 | node2     | top_group       | RAFT_FOLLOWER |
1148549230
```

Raft consensus is working correctly if all of these 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 starts a new election to define a new `RAFT_LEADER` . During an election, there might be an intermediary situation where there's no `RAFT_LEADER` , and some of the nodes consider themselves as `RAFT_CANDIDATE` . The whole election can't take longer than `bdr.raft_global_election_timeout` (by default it's set to 6 seconds). If the query above returns an in-election situation, then wait for `bdr.raft_global_election_timeout` , and run the query again. If after `bdr.raft_global_election_timeout` has passed and some the listed conditions are still not met, then Raft consensus isn't working.

Raft consensus might not be working correctly on only a single node. For example, one of the nodes doesn't recognize the current leader and considers itself as a `RAFT_CANDIDATE` . In this case, it's important to make sure that:

- All PGD nodes are accessible to each other through both regular and replication connections (check file `pg_hba.conf`).
- PGD versions are the same on all nodes.
- `bdr.raft_global_election_timeout` is the same on all nodes.

In some cases, especially if nodes are geographically distant from each other or network latency is high, the default value of `bdr.raft_global_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_global_election_timeout` to 30 seconds on all nodes. For PGD 3.6.11 and later, setting `bdr.raft_global_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, monitoring alert levels are defined as follows:

- status=UNKNOWN, message=This node is not part of any PGD 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();
node_group_name | status |
message
----------------/--------+---------------------------------------
mygroup         | OK     | Raft Consensus is working
correctly
```

Two further views that can give a finer-grained look at the state of Raft consensus are `bdr.stat_raft_state` , which provides the state of the Raft consensus on the local node, and `bdr.stat_raft_followers_state` , which provides a view when on the Raft leader (it is empty on other nodes) regarding the state of the followers of that Raft leader.

Monitoring replication slots

Each PGD node keeps:

- One replication slot per active PGD 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 PGD group slot name follows the convention `bdr_<DATABASE>_<GROUP>`. You can access the group slot using the function `bdr.local_group_slot_name()`.

Peer replication slots must be active on all nodes at all times. If a peer replication slot isn't active, then it might mean either:

- The corresponding peer is shut down or not accessible.
- PGD replication is broken.

Grep the log file for `ERROR` or `FATAL`, and also check `bdr.event_summary` 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 PGD group replication slot is, however, inactive most of the time. PGD maintains this slot and advances its LSN when all other peers already consumed the corresponding transactions. Consequently, it's not necessary to monitor the status of the group slot.

The function `bdr.monitor_local_replslots()` provides a summary of whether all PGD node replication slots are working as expected. This summary is also available on subscriber-only nodes that are operating as subscriber-only group leaders in a PGD cluster when optimized topology is enabled. For example:

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

One of the following status summaries is returned:

Status	Message
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, PGD transactions are committed only to the local node. In that case, a transaction's `COMMIT` is processed quickly.

PGD's Commit Scopes feature offers a range of synchronous transaction commit scopes that allow you to balance durability, consistency, and performance for your particular queries. You can monitor these transactions by examining the `bdr.stat_activity` catalog. The processes report different `wait_event` states as a transaction is committed. This monitoring only covers transactions in progress and doesn't provide historical timing information.

18 Testing and tuning PGD clusters

You can test PGD applications using the following approaches:

- Trusted Postgres Architect
- pgd_bench with CAMO/failover options

Trusted Postgres Architect

Trusted Postgres Architect is the system used by EDB to deploy reference architectures, including those based on EDB Postgres Distributed.

Trusted Postgres Architect 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 Trusted Postgres Architect tests that verify the main expected properties of the application.

pgd_bench

The Postgres benchmarking application `pgbench` was extended in PGD 5.0 in the form of a new application: pgd_bench.

pgd_bench is a regular command-line utility that's added to the PostgreSQL bin directory. The utility is based on the PostgreSQL pgbench tool but supports benchmarking CAMO transactions and PGD-specific workloads.

Functionality of pgd_bench is a superset of pgbench functionality but requires the BDR extension to be installed to work properly.

Key differences include:

- Adjustments to the initialization (`-i` flag) with the standard pgbench scenario to prevent global lock timeouts in certain cases.
- `VACUUM` command in the standard scenario is executed on all nodes.
- pgd_bench releases are tied to the releases of the BDR extension and are built against the corresponding Postgres distribution. This information is reflected in the output of the `--version` flag.

The current version allows you to run failover tests while using CAMO or regular PGD deployments.

The following options were added:

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

- Use `-m camo` or `-m failover` to specify the mode for pgd_bench. You can use the `-m failover` specification to test failover in regular PGD deployments.

```
--retry
retry transactions on failover
```

- 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.

In addition to these options, you must specify the connection information about the peer node for failover inDSN form.

Here's an example in a CAMO environment:

```
    pgd_bench -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 pgd_bench 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 you specify `--retry`, then in-flight transactions are retried. In this scenario, there's no way to find the status of in-flight transactions.

Notes on pgd_bench usage

- When using custom init-scripts, it's important to understand implications behind the DDL commands. We generally recommend waiting for the secondary nodes to catch up on the data-load steps before proceeding with DDL operations such as `CREATE INDEX`. The latter acquire global locks that can't be acquired until the data load is complete and thus might time out.

- No extra steps are taken to suppress client messages, such as `NOTICE` and `WARNING` messages emitted by PostgreSQL and or any possible extensions, including the BDR extension. It's your responsibility to suppress them by setting appropriate variables, such as `client_min_messages`, `bdr.camo_enable_client_warnings`, and so on.

Performance testing and tuning

PGD allows you to issue write transactions onto multiple nodes. Bringing those writes back together onto each node has a performance cost.

First, replaying changes from another node has a CPU cost and an I/O cost, and it generates WAL records. The resource use is usually less than in the original transaction since CPU overhead is 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. pgd_bench allows you to write custom test scripts specific to your use case so you can understand the overhead of your SQL and measure the impact of concurrent execution.

If PGD is running slow, then we suggest the following:

1. Write a custom test script for pgd_bench, 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 occur in production, using the same number of sessions in total as you did on one node. This technique 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.

19 Upgrading

While PGD and Postgres are closely related, they're separate products with separate upgrade paths. This section covers how to upgrade both PGD and Postgres.

Upgrading PGD

EDB Postgres Distributed is a flexible platform. This means that your upgrade path depends largely on how you installed PGD.

- Upgrading with TPA — If you installed using TPA, you can use its automated upgrade feature to upgrade to the latest minor versions.
- Upgrading manually — If you manually installed and configured your PGD cluster, you can move a cluster between versions, both minor and major.
- Upgrade paths — Several supported upgrade paths are available.
- Compatibility changes — If you're upgrading from PGD 3.x or 4.x to PGD 5.x or later, you need to understand the compatibility changes between versions.

Upgrading Postgres

- In-place Postgres upgrades — How to use `bdr_pg_upgrade` to manually upgrade the Postgres version on a node, or nodes.
- Rolling major version upgrades — How to perform a major version upgrade of Postgres on a cluster deployed using TPA.

Other upgrades

- Application schema upgrades — A guide for safely upgrading your application's schema when running multiple distributed servers with PGD.

19.1 Upgrading PGD clusters with TPA

> **No Postgres major version upgrades**
>
> TPA doesn't currently support major version upgrades of Postgres.
>
> To perform a major version upgrade of Postgres, see In-place Postgres major version upgrades.

If you used TPA to install your cluster, you can also use TPA to upgrade it. The techniques outlined here can perform minor and major version upgrades of the PGD software. They can also perform minor version upgrades of Postgres.

You can read more about the capabilities of TPA upgrades in Upgrading your cluster in the TPA documentation.

> **Always test first**
>
> If possible, always test upgrade processes in a QA environment first. This helps to ensure that there are no unexpected factors to take into account. TPA's ability to reproducibly deploy a PGD configuration makes it much easier to build a test environment to work with.

Minor and major PGD upgrades

TPA automatically manages minor version upgrades of PGD.

Major version upgrades of PGD require changes to the TPA `config.yml` file, which contains the deployment configuration.

When upgrading to PGD 5 from previous PGD major versions, you can use `tpaexec reconfigure`. This command helps you make appropriate modifications to your deployment configuration.

The `reconfigure` command requires settings for architecture (the only supported setting is `PGD_Always_ON`) and PGD Proxy routing (`--pgd-proxy-routing <global|local>`) to run. Remember to back up your deployment configuration before running, and use the `--describe` and `--output` options to preview the reconfiguration.

Prerequisites

- You need the cluster configuration directory created when TPA deployed your PGD cluster.
- If performing a major version upgrade of PGD, ensure that `tpaexec reconfigure` was run and appropriate configuration changes were made.

Upgrading

Run:

```
tpaexec upgrade clustername
```

Where `clustername` is the name of the cluster and the path to the cluster configuration directory. By default, TPA upgrades each node of the cluster to the latest minor versions of the software the nodes were configured with.

TPA's automated rolling upgrade procedure

TPA first tests the cluster and then the nodes.

Each node is then isolated from the cluster, upgraded, and returned to operation within the cluster.

TPA upgrades - step by step

- Checks that all preconditions for upgrading the cluster are met.
- For each instance in the cluster:
    - Checks that it has the correct repositories configured.
        - Checks that the required Postgres packages are available in those repositories.
    - For each BDR node in the cluster, one at a time:
        - Fences the node off to ensure that pgd-proxy doesn't send any connections to it.
        - Stops, updates, and restarts Postgres.
        - Unfences the node so it can receive connections again.
        - Updates pgbouncer, pgd-proxy, and pgd-cli, as applicable for this node.

19.2 Upgrading PGD clusters manually

Because EDB Postgres Distributed consists of multiple software components, the upgrade strategy depends partially on the components that are being upgraded.

In general, you can upgrade the cluster with almost zero downtime by using an approach called *rolling upgrade*. Using this approach, nodes are upgraded one by one, and the application connections are switched over to already upgraded nodes.

You can also stop all nodes, perform the upgrade on all nodes, and only then restart the entire cluster. This approach is the same as 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. However, it incurs downtime and we don't recommend it unless you can't perform the rolling upgrade for some reason.

To upgrade an EDB Postgres Distributed cluster:

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:

- Upgrade nodes in place to the newer software version. See Rolling server software upgrades.
- Replace nodes with ones that have the newer version installed. See Rolling upgrade using node join.

You can use both of these approaches 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, suppose nodeA has PGD 3.7.16, while nodeB and nodeC have 4.1.0. In this state, the replication and group management uses the protocol and features from the oldest version (3.7.16 in this example), so any new features provided by the newer version that require changes in the protocol are disabled. Once all nodes are upgraded to the same version, the new features are enabled.

Similarly, when a cluster with WAL-decoder-enabled nodes is going through a rolling upgrade, WAL decoder on a higher version of PGD node produces logical change records (LCRs) with a higher pglogical version. WAL decoder on a lower version of PGD node produces LCRs with a lower pglogical version. As a result, WAL senders on a higher version of PGD nodes aren't expected to use LCRs due to a mismatch in protocol versions. On a lower version of PGD nodes, WAL senders can continue to use LCRs. Once all the PGD nodes are on the same PGD version, WAL senders use LCRs.

A rolling upgrade starts with a cluster with all nodes at a prior release. It then proceeds by upgrading one node at a time to the newer release, until all nodes are at the newer release. There must be no more than two versions of the software running at the same time. An upgrade must be completed, with all nodes fully upgraded, before starting another upgrade.

An upgrade process can take more time when caution is required to reduce business risk. However, we don't recommend running mixed versions of the software indefinitely.

While you can use a rolling upgrade for upgrading a major version of the software, we don't support mixing PostgreSQL, EDB Postgres Extended, and EDB Postgres Advanced Server in one cluster. So you can't use this approach to change the Postgres variant.

> **Warning**
>
> Downgrades of EDB Postgres Distributed aren't supported. They require that you manually rebuild the cluster.

Rolling server software upgrades

A rolling upgrade is where the server software upgrade is upgraded sequentially on each node in a cluster without stopping the cluster. Each node is temporarily stopped from participating in the cluster and its server software is upgraded. Once updated, it's returned to the cluster, and it then catches up with the cluster's activity during its absence.

The actual procedure depends on whether the Postgres component is being upgraded to a new major version.

During the upgrade process, you can switch the application over to a node that's currently not being upgraded to provide continuous availability of the database for applications.

Rolling upgrade using node join

The other method to upgrade 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, a potentially large data transfer is required.

Take care not to use features that are available only in the newer Postgres version until all nodes are upgraded to the newer and same release of Postgres. This is especially true for any new DDL syntax that was added to a newer release of Postgres.

> **Note**
>
> `bdr_init_physical` makes a byte-by-byte copy of the source node so you can't use it while upgrading from one major Postgres version to another. In fact, currently `bdr_init_physical` requires that even the PGD version of the source and the joining node be exactly the same. You can't use it for rolling upgrades by way of joining a new node method. Instead, use a logical join.

Upgrading a CAMO-enabled cluster

Upgrading a CAMO-enabled cluster requires upgrading CAMO groups one by one while disabling the CAMO protection for the group being upgraded and reconfiguring it using the new commit scope-based settings.

We recommended the following approach for upgrading two BDR nodes that constitute a CAMO pair to PGD 5.0:

1. Ensure `bdr.enable_camo` remains `off` for transactions on any of the two nodes, or redirect clients away from the two nodes. Removing the CAMO pairing while attempting to use CAMO leads to errors and prevents further transactions.
2. Uncouple the pair by deconfiguring CAMO either by resetting `bdr.camo_origin_for` and `bdr.camo_parter_of` (when upgrading from BDR 3.7.x) or by using `bdr.remove_camo_pair` (on BDR 4.x).
3. Upgrade the two nodes to PGD 5.0.
4. Create a dedicated node group for the two nodes and move them into that node group.
5. Create a commit scope for this node group and thus the pair of nodes to use CAMO.
6. Reactivate CAMO protection again either by setting a `default_commit_scope` or by changing the clients to explicitly set `bdr.commit_scope` instead of `bdr.enable_camo` for their sessions or transactions.
7. If necessary, allow clients to connect to the CAMO-protected nodes again.

Upgrade preparation

Each major release of the software contains several changes that might affect compatibility with previous releases. These might affect the Postgres configuration, deployment scripts, as well as applications using PGD. We recommend considering these changes and making any needed adjustments in advance of the upgrade.

See individual changes mentioned in the release notes and any version-specific upgrade notes.

Server software upgrade

Upgrading EDB Postgres Distributed on individual nodes happens in place. You don't need to back up and restore when upgrading the BDR extension.

BDR extension upgrade

The BDR extension upgrade process consists of a few steps.

Stop Postgres

During the upgrade of binary packages, it's usually best to stop the running Postgres server first. Doing so ensures that mixed versions don't get loaded in case of an unexpected restart during the upgrade.

Upgrade packages

The first step in the upgrade is to install the new version of the BDR packages. This installation installs both the new binary and the extension SQL script. This step is specific to the operating system.

Start Postgres

Once packages are upgraded, you can start the Postgres instance. The BDR extension is upgraded upon start when the new binaries detect the older version of the extension.

Postgres upgrade

The process of in-place upgrade of Postgres depends on whether you're upgrading to a new minor version of Postgres or to a 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 more steps, like reindexing, might be recommended for specific minor version upgrades. Refer to the release notes of the version of Postgres you're upgrading to.

Major version Postgres upgrade

Upgrading to a new major version of Postgres is more complicated than upgrading to a minor version.

EDB Postgres Distributed provides a bdr_pg_upgrade command line utility, which you can use to do in-place Postgres major version upgrades.

> **Note**
>
> When upgrading to a new major version of any software, including Postgres, the BDR extension, and others, it's always important to ensure your application is compatible with the target version of the software you're upgrading.

Upgrade check and validation

After you upgrade your PGD node, you can verify the current version of the binary:

```
SELECT bdr.bdr_version();
```

Always check your monitoring after upgrading a node to confirm that the upgraded node is working as expected.

Moving from HARP to PGD Proxy

HARP can temporarily coexist with the new connection management configuration. This means you can:

- Upgrade a whole pre-5 cluster to a PGD 5 cluster.
- Set up the connection routing.
- Replace HARP Proxy with PGD Proxy.
- Move application connections to PGD Proxy instances.
- Remove the HARP Manager from all servers.

We strongly recommend doing this as soon as possible after upgrading nodes to PGD 5. HARP isn't certified for long-term use with PGD 5.

TPA provides some useful tools for this and will eventually provide a single-command upgrade path between PGD 4 and PGD 5.

19.3 Supported PGD upgrade paths

Upgrading within version 5

You can upgrade from any version 5.x release to a later 5.x release.

Upgrading from version 4 to version 5

Upgrades from PGD 4 to PGD 5 are supported from version 4.3.0. For earlier versions, upgrade to 4.3.0 before upgrading to 5. See Upgrading within 4 for more information.

Generally, we recommend that you upgrade to the latest version 4 release before upgrading to the latest version 5 release. After upgrading to 4.3.0 or later, the following upgrade paths are possible.

From version	To version
4.3.0	5.0.0 or later
4.3.1	5.1.0 or later
4.3.2	5.1.0 or later
4.3.3	5.1.0 or later

Upgrading from version 3.7 to version 5

Starting with version 3.7.23, you can upgrade directly to version 5.3.0 or later. For earlier versions, upgrade to 3.7.23 before upgrading to 5. See Upgrading within version 3.7 for more information.

19.4 Compatibility changes

Many changes in PGD 5 aren't backward compatible with PGD 4 or PGD 3.7.

Connection routing

HARP Manager doesn't exist anymore. It's been replaced by new connection management configuration.

HARP Proxy is replaced by similarly functioning PGD Proxy, which removes any deprecated features and is configured through connection management configuration.

Commit At Most Once

CAMO configuration is now done through commit scopes. The `bdr.camo_pairs` catalog and any related manipulation functions don't exist anymore. The `bdr.enable_camo` GUC was removed. The `synchronous_replication_availability` GUC doesn't affect CAMO anymore. Use the `DEGRADE ON ... TO ASYNC` clause of a commit scope.

Eager All-Node Replication

The `global` scope no longer exists. To create scope with the same behavior, use Group Commit.

```
SELECT
bdr.create_commit_scope(
    commit_scope_name := 'eager_scope',
    origin_node_group := 'top_group',
    rule := 'ALL (top_group) GROUP COMMIT (conflict_resolution = eager, commit_decision = raft) ABORT ON (timeout
= 60s)',
    wait_for_ready :=
true
);
```

The `bdr.global_commit_timeout` GUC was removed. Use the `ABORT ON` clause for the commit scope.

Lag Control

Similarly to CAMO and Eager, Lag Control configuration was also moved to commit scopes for more flexible durability configuration.

Catalogs

- `bdr.workers` doesn't show worker-specific info like `worker_commit_timestamp` anymore.
- `bdr.worker_errors` is deprecated and lost most of the info.
- `bdr.state_journal_details` is deprecated and lost most of the info.
- `bdr.event_summary` replaces `bdr.worker_errors` and `bdr.state_journal_details` with additional info like Raft role changes.
- The table `bdr.node_catchup_info` now has the user-consumable view `bdr.node_catchup_info_details`, which shows info in a more friendly way.
- Witness node is no longer distinguished by the replication sets it replicates but by using the `node_kind` value in `bdr.node_summary`.

- All the Raft (consensus) related tables and functions were adjusted to support multiple Raft instances (sub-group Raft).
- `bdr.node_pre_commit` view and the underlying table was removed, as the information is no longer stored in a table.
- `bdr.commit_decisions` view was added and replaces the `bdr.node_pre_commit` one.
- Multiple internal autopartition tables were replaced by taskmgr ones, as the mechanism behind it was generalized.
- `bdr.network_monitoring` view was removed along with underlying tables and functions.
- Many catalogs were added and some have new columns, as described in Catalogs. These aren't breaking changes strictly speaking, but we recommend reviewing them when upgrading.

19.5 In-place Postgres major version upgrades

You can upgrade a PGD node to a newer major version of Postgres using the command-line utility bdr_pg_upgrade.

bdr_pg_upgrade is a wrapper around the standard pg_upgrade that adds PGD-specific logic to the process to ensure a smooth upgrade.

Terminology

This terminology is used when describing the upgrade process and components involved:

Postgres cluster — The database files, both executables and data, that make up a Postgres database instance on a system when run.

Old Postgres cluster — The existing Postgres cluster to upgrade, the one from which to migrate data.

New Postgres cluster — The new Postgres cluster that data is migrated to. This Postgres cluster must be one major version ahead of the old cluster.

Precautions

Standard Postgres major version upgrade precautions apply, including the fact both Postgres clusters must meet all the requirements for pg_upgrade.

Additionally, don't use bdr_pg_upgrade if other tools are using replication slots and replication origins. Only PGD slots and origins are restored after the upgrade.

You must meet several prerequisites for bdr_pg_upgrade:

- Disconnect applications using the old Postgres cluster. You can, for example, redirect them to another node in the PGD cluster.
- Configure peer authentication for both Postgres clusters. bdr_pg_upgrade requires peer authentication.
- The same PGD version must be installed on both clusters.
- The PGD version must be 4.1.0 or later. Version 3.7.22 and later is also supported.
- The new cluster must be in a shutdown state.
- You must install PGD packages in the new cluster.
- The new cluster must already be initialized and configured as needed to match the old cluster configuration.
- Databases, tables, and other objects must not exist in the new cluster.

We also recommend having the old Postgres cluster up prior to running bdr_pg_upgrade. The CLI starts the old Postgres cluster if it's shut down.

Usage

To upgrade to a newer major version of Postgres, you must first install the new version.

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, you can pass the following parameters to bdr_pg_upgrade.

Required parameters

Specify these parameters either in the command line or, for all but the `--database` parameter, in their equivalent environment variable. They're used by bdr_pg_upgrade.

- `-b, --old-bindir` — Old Postgres cluster bin directory.
- `-B, --new-bindir` — New Postgres cluster bin directory.
- `-d, --old-datadir` — Old Postgres cluster data directory.
- `-D, --new-datadir` — New Postgres cluster data directory.
- `--database` — PGD database name.

Optional parameters

These parameters are optional and are used by bdr_pg_upgrade:

- `-p, --old-port` — Old cluster port number.
- `-s, --socketdir` — Directory to use for postmaster sockets during upgrade.
- `--check` — Specify to only perform checks and not modify clusters.

Other parameters

Any other parameter that's not one of the above is passed to pg_upgrade. pg_upgrade accepts the following parameters:

- `-j, --jobs` — Number of simultaneous processes or threads to use.
- `-k, --link` — Use hard links instead of copying files to the new cluster.
- `-o, --old-options` — Option to pass to old postgres command. Multiple invocations are appended.
- `-O, --new-options` — Option to pass to new postgres command. Multiple invocations are appended.
- `-N, --no-sync` — Don't wait for all files in the upgraded cluster to be written to disk.
- `-P, --new-port` — New cluster port number.
- `-r, --retain` — Retain SQL and log files even after successful completion.
- `-U, --username` — Cluster's install user name.
- `--clone` — Use efficient file cloning.

Environment variables

You can use these environment variables in place of command-line parameters:

- `PGBINOLD` — Old Postgres cluster bin directory.
- `PGBINNEW` — New Postgres cluster bin directory.
- `PGDATAOLD` — Old Postgres cluster data directory.
- `PGDATANEW` — New Postgres cluster data directory.
- `PGPORTOLD` — Old Postgres cluster port number.
- `PGSOCKETDIR` — Directory to use for postmaster sockets during upgrade.

Example

Given a scenario where:

- Old Postgres cluster bin directory is `/usr/lib/postgresql/13/bin`.
- New Postgres cluster bin directory is `/usr/lib/postgresql/14/bin`.
- Old Postgres cluster data directory is `/var/lib/postgresql/13/main`.
- New Postgres cluster data directory is `/var/lib/postgresql/14/main`.
- Database name is `bdrdb`.

You can use the following command 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

These steps are performed when running bdr_pg_upgrade.

> **Note**
>
> When `--check` is supplied as an argument to bdr_pg_upgrade, the CLI skips steps that modify the database.

PGD Postgres checks

Steps	`--check` supplied
Collecting pre-upgrade new cluster control data	`run`
Checking new cluster state is shutdown	`run`
Checking PGD 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`
Waiting for all slots to be flushed	`skip`
Disconnecting from old cluster	`skip`
Stopping old cluster	`skip`
Starting old cluster with PGD 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**
>
> If supplied, `--check` is passed to pg_upgrade.

PGD 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 PGD disabled	`skip`
Connecting to new cluster	`skip`
Creating replication origin, repeated for each origin	`skip`
Advancing replication origin, repeated for each origin	`skip`
Creating replication slot, repeated for each slot	`skip`
Stopping new cluster	`skip`

19.6 Application schema upgrades

Similar to the upgrade of EDB Postgres Distributed, 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. This approach imposes some downtime.

To eliminate this downtime, EDB Postgres Distributed offers useful tools to perform a rolling application schema upgrade.

The following recommendations and tips reduce the impact of the application schema upgrade on the cluster.

Rolling application schema upgrades

By default, DDL is automatically sent to all nodes. You can control this behavior manually, as described in DDL replication. You can use this approach to create differences between database schemas across nodes.

PGD 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.

> **Warning**
>
> You must manage rolling application schema upgrades outside of PGD.
>
> Careful scripting is required to make this work correctly on production clusters. We recommend extensive testing.

See Replicating between nodes with differences for details.

When one node runs DDL that adds a new table, nodes that haven't 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. Perform this configuration before adding tables to any node. This setting is intended to be permanent.

Execute the following query **separately on each node**. Replace `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 haven't 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. Perform this before adding columns to one node. This setting is intended to be permanent.

Execute the following query **separately on each node**. Replace `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 haven't yet received the latest DDL need to handle the missing column. This situation causes a `source_column_missing` conflict, which uses the `use_default_value` resolver. Thus, columns that don't accept NULLs and don't have a DEFAULT value require a two-step process:

1. Remove the NOT NULL constraint, or add a DEFAULT value for a column on all nodes.
2. Remove the column.

You can remove constraints in a rolling manner. There's 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 might not rewrite the underlying table data. In that case, it's only a metadata update of the underlying column type. Rewriting a table is normally restricted. However, 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 non-replicated environment on all the nodes, one by one. See ALTER TABLE for more details.

19.7 Performing a Postgres major version rolling upgrade on a PGD cluster built with TPA

Upgrading Postgres major versions

Upgrading a Postgres database's major version to access improved features, performance enhancements, and security updates is a common administration task. Doing the same for an EDB Postgres Distributed (PGD) cluster deployed with Trusted Postgres Architect (TPA) is essentially the same process but performed as a rolling upgrade.

The rolling upgrade process allows updating individual cluster nodes to a new major Postgres version while maintaining cluster availability and operational continuity. This approach minimizes downtime and ensures data integrity by allowing the rest of the cluster to remain operational as each node is upgraded sequentially.

The following overview of the general instructions and worked example help to provide a smooth and controlled upgrade process.

Prepare the upgrade

To prepare for the upgrade, identify the subgroups and nodes you're trying to upgrade and note an initial upgrade order.

To do this, connect to one of the nodes using SSH and run the `pgd show-nodes` command:

```
sudo -u postgres pgd show-
nodes
```

The `pgd show-nodes` command shows you all the nodes in your PGD cluster and the subgroup to which each node belongs. Then you want to find out which node is the write leader in each subgroup:

```
sudo -u postgres pgd show-
groups
```

This command outputs a list of the different groups/subgroups running in your cluster and the write leader of each group. To maintain operational continuity, you need to switch write leaders over to another node in their subgroup before you can upgrade them. To keep the number of planned switchovers to a minimum, when upgrading a subgroup of nodes, upgrade the writer leaders last.

Even though you verified which node is the current write leader for planning purposes, the write leader of a subgroup could change to another node at any moment for operational reasons before you upgrade that node. Therefore, you still need to verify that a node isn't the write leader just before upgrading that node.

You now have enough information to determine your upgrade order, one subgroup at a time, aiming to upgrade the identified write leader node last in each subgroup.

Perform the upgrade on each node

> **Note**
>
> To help prevent data loss, before starting the upgrade process, ensure that your databases and configuration files are backed up.

Using the preliminary order, perform the following steps on each node while connected via SSH:

- **Confirm the current Postgres version**
    - View versions from PGD:

      `sudo -u postgres pgd show-version`.
    - Ensure that the expected major version is running.

- **Verify that the target node isn't the write leader**
  - Check whether the target node is the write leader for the group you're upgrading:

    ```
    sudo -u postgres pgd show-groups
    ```

  - If the target node is the current write leader for the group/subgroup you're upgrading, perform a planned switchover to another node:

    ```
    sudo -u postgres pgd switchover --group-name <group_name> --node-name <new_write_leader_node_name>
    ```

- **Stop Postgres on the target node**
  - Stop the Postgres service on the current node:

    ```
    sudo systemctl stop postgres
    ```

    The target node is no longer actively participating as a node in the cluster.

- **Install PGD and utilities**
  - Install PGD and its utilities compatible with the Postgres version you're upgrading to:

    ```
    sudo apt install edb-bdr<new_pgd_version_number>-pg<new_postgres_version_number> edb-bdr-utilities
    ```

- **Initialize the new Postgres instance**
  - Create a directory to house the database files for the new version of PostgreSQL:

    ```
    sudo mkdir -p /opt/postgres/datanew
    ```

  - Ensure that the user postgres has ownership permissions to the directory using `chown`.
  - Initialize a new PostgreSQL database cluster in the directory you just created. This step involves using the `initdb` command provided by the newly installed version of PostgreSQL. Include the `--data-checksums` flag to ensure the cluster uses data checksums.

    ```
    sudo -u postgres <path_to_postgres_bin>/initdb -D /opt/postgres/datanew --data-checksums
    ```

    Replace `<path_to_postgres_bin>` with the path to the bin directory of the newly installed PostgreSQL version.

    You may need to run this command as the postgres user or another user with appropriate permissions.

- **Migrate configuration to the new Postgres version**
  - Locate the following configuration files in your current PostgreSQL data directory:
    - `postgresql.conf` — The main configuration file containing settings related to the database system.
    - `postgresql.auto.conf` — Contains settings set by PostgreSQL, such as those modified by the `ALTER SYSTEM` command.
    - `pg_hba.conf` — Manages client authentication, specifying which users can connect to which databases from which hosts.
    - The entire `conf.d` directory (if present) — Allows for organizing configuration settings into separate files for better manageability.
  - Copy these files and the `conf.d` directory to the new data directory you created for the upgraded version of PostgreSQL.

- **Verify the Postgres service is inactive**
  - Before proceeding, it's important to ensure that no PostgreSQL processes are active for both the old and the new data directories. This verification step prevents any data corruption or conflicts during the upgrade process.

    Use the `sudo systemctl status postgres` command to verify that Postgres was stopped. If it isn't stopped, run `systemctl stop postgres` and verify again that it was stopped.

- **Swap PGDATA directories for version upgrade**
  - Rename `/opt/postgres/data` to `/opt/postgres/dataold` and `/opt/postgres/datanew` to `/opt/postgres/data`.

  This step readies your system for the next crucial phase: running pg_upgrade to finalize the PostgreSQL version transition.

- **Verify upgrade feasibility**
  - The `bdr_pg_upgrade` tool offers a `--check` option designed to perform a preliminary scan of your current setup, identifying any potential issues that could hinder the upgrade process.

    You need to run this check from an upgrade directory with ownership given to user postgres, such as `/home/upgrade/`, so that the upgrade log files created by `bdr_pg_upgrade` can be stored. To initiate the safety check, append the `--check` option to your `bdr_pg_upgrade` command.

    This operation simulates the upgrade process without making any changes, providing insights into any compatibility issues, deprecated features, or configuration adjustments required for a successful upgrade.

  - Address any warnings or errors indicated by this check to ensure an uneventful transition to the new version.

- **Execute the Postgres major version upgrade**
  - Execute the upgrade process by running the `bdr_pg_upgrade` command without the `--check` option.
  - It's essential to monitor the command output for any errors or warnings that require attention.
  - The time the upgrade process take depends on the size of your database and the complexity of your setup.

- **Update the Postgres service configuration**
  - Update the service configuration to reflect the new PostgreSQL version by updating the version number in the `postgres.service` file:

    ```
    sudo sed -i -e 's/<old_version_number>/<new_version_number>/g' /etc/systemd/system/postgres.service
    ```

  - Refresh the system's service manager to apply these changes:

    ```
    sudo systemctl daemon-reload
    ```

- **Restart Postgres**
  - Proceed to restart the PostgreSQL service:

    ```
    systemctl start postgres
    ```

- **Validate the new Postgres version**
  - Verify that your PostgreSQL instance is now upgraded:

    ```
    sudo -u postgres pgd show-version
    ```

- **Clean up post-upgrade**
  - Run `vacuumdb` with the `ANALYZE` option immediately after the upgrade but before introducing a heavy production load. Running this command minimizes the immediate performance impact, preparing the database for more accurate testing.
  - Remove the old version's data directory, `/opt/postgres/dataold`.

Reconcile the upgrade with TPA

TPA needs to continue to manage the deployment effectively after all the nodes have been upgraded. Therefore, it's necessary to reconcile the upgraded nodes with TPA.

Follow these steps to update the configuration and redeploy the PGD cluster through TPA.

- **Update the `config.yml`**
    - Change the `config.yml` of the TPA-managed cluster to the new version:

      ```
      cluster_vars: postgres_version: '<new_version_number>'
      ```

- **Use `tpaexec` to redeploy the PGD cluster with the updated `config.yml`**
    - Use this the `deploy` option:

      ```
      tpaexec deploy <cluster_name>
      ```

The worked example that follows shows upgrading the Postgres major version from 15 to 16 on a PGD 5 cluster deployed with TPA in detail.

Worked example

This worked example starts with a TPA-managed PGD cluster deployed using the AWS quick start. The cluster has three nodes: kaboom, kaolin, and kaftan, all running Postgres 15.

This example starts with kaboom.

> **Note**
>
> Some steps of this process involve running commands as the Postgres owner. We refer to this user as postgres throughout, when appropriate. If you're running EDB Postgres Advanced Server, substitute the postgres user with enterprisedb in all relevant commands.

Confirm the current Postgres version

SSH into kaboom to confirm the major version of Postgres is expected:

```
sudo -u postgres pgd show-
version
```

The output will be similar to this for your cluster:

```
 Node   BDR Version Postgres Version
----   ----------- ----------------
kaboom 5.4.0       15.6 (Debian 15.6-2EDB.buster)
kaftan 5.4.0       15.6 (Debian 15.6-2EDB.buster)
kaolin 5.4.0       15.6 (Debian 15.6-2EDB.buster)
```

Confirm that the Postgres version is the expected version.

Verify that the target node isn't the write leader

The cluster must be available throughout the process (that is, a *rolling* upgrade). There must always be an available write leader to maintain continuous cluster availability. So, if the target node is the current write leader, you must perform a planned switchover of the write leader node before upgrading it so that a write leader is always available.

While connected via SSH to kaboom, see which node is the current write leader of the group you're upgrading using the `pgd show-groups` command:

```
sudo -u postgres pgd show-
groups
```

In this case, you can see that kaboom is the current write leader of the sole subgroup `dc1_subgroup`:

```
Group        Group ID   Type   Parent Group Location Raft Routing Write Leader
-----        --------   ----   ------------ -------- ---- ------- ------------
democluster  1935823863 global                       true false
dc1_subgroup 1302278103 data   democluster  dc1      true true    kaboom
```

So you must perform a planned switchover of the write leader of `dc1_subgroup` to another node in the cluster.

Perform a planned switchover

Change the write leader to kaftan so kaboom's Postgres instance can be stopped:

```
sudo -u postgres pgd switchover --group-name dc1_subgroup --node-name
kaftan
```

After the switchover is successful, it's safe to stop Postgres on the target node. Of course, if kaftan is still the write leader when you come to upgrading it, you'll need to perform another planned switchover at that time.

Stop Postgres on the target node

While connected via SSH to the target node (in this case, kaboom), stop Postgres on the node by running:

```
sudo systemctl stop
postgres
```

This command halts the server on kaboom. Your cluster continues running using the other two nodes.

Install PGD and utilities

Next, install the new version of Postgres (PG16) and the upgrade tool:

```
sudo apt install edb-bdr5-pg16 edb-bdr-
utilities
```

Initialize the new Postgres instance

Make a new data directory for the upgraded Postgres, and give the postgres user ownership of the directory:

```
sudo mkdir
/opt/postgres/datanew
sudo chown -R postgres:postgres
/opt/postgres/datanew
```

Then, initialize Postgres 16 in the new directory:

```
sudo -u postgres /usr/lib/postgresql/16/bin/initdb
\
  -D /opt/postgres/datanew
\
  -E UTF8 \
  --lc-collate=en_US.UTF-8 \
  --lc-ctype=en_US.UTF-8 \
  --data-checksums
```

This command creates a PG16 data directory for configuration, `/opt/postgres/datanew`.

Migrate configuration to the new Postgres version

The next step copies the configuration files from the old Postgres version (PG15) to the new Postgres version's (PG16). Configuration files reside in each version's data directory.

Copy over the `postgresql.conf`, `postgresql.auto.conf`, and `pg_hba.conf` files and the whole `conf.d` directory:

```
sudo -u postgres cp /opt/postgres/data/postgresql.conf
/opt/postgres/datanew/
sudo -u postgres cp /opt/postgres/data/postgresql.auto.conf
/opt/postgres/datanew/
sudo -u postgres cp /opt/postgres/data/pg_hba.conf
/opt/postgres/datanew/
sudo -u postgres cp -r /opt/postgres/data/conf.d/
/opt/postgres/datanew/
```

Verify the Postgres service is inactive

Although you previously stopped the Postgres service on the target node, kaboom, to verify it's stopped, run the `systemctl status postgres` command:

```
sudo systemctl status
postgres
```

The output of the `status` command shows that the Postgres service has stopped running:

```
● postgres.service - Postgres 15 (TPA)
   Loaded: loaded (/etc/systemd/system/postgres.service; enabled; vendor preset: enabled)
   Active: inactive (dead) since Wed 2024-03-20 15:32:18 UTC; 4min 9s ago
 Main PID: 24396 (code=exited, status=0/SUCCESS)

Mar 20 15:32:18 kaboom postgres[25032]: [22-1] 2024-03-20 15:32:18 UTC
[pgdproxy@10.33.125.89(20108)/[unknown]/bdrdb:25032]: [1] FA
Mar 20 15:32:18 kaboom postgres[25033]: [22-1] 2024-03-20 15:32:18 UTC
[pgdproxy@10.33.125.89(20124)/[unknown]/bdrdb:25033]: [1] FA
Mar 20 15:32:18 kaboom postgres[25034]: [22-1] 2024-03-20 15:32:18 UTC
[pgdproxy@10.33.125.88(43534)/[unknown]/bdrdb:25034]: [1] FA
Mar 20 15:32:18 kaboom postgres[25035]: [22-1] 2024-03-20 15:32:18 UTC
[pgdproxy@10.33.125.88(43538)/[unknown]/bdrdb:25035]: [1] FA
Mar 20 15:32:18 kaboom postgres[25036]: [22-1] 2024-03-20 15:32:18 UTC
[pgdproxy@10.33.125.87(37292)/[unknown]/bdrdb:25036]: [1] FA
Mar 20 15:32:18 kaboom postgres[25037]: [22-1] 2024-03-20 15:32:18 UTC
[pgdproxy@10.33.125.87(37308)/[unknown]/bdrdb:25037]: [1] FA
Mar 20 15:32:18 kaboom postgres[24398]: [24-1] 2024-03-20 15:32:18 UTC [@//:24398]: [15] LOG:  checkpoint
complete: wrote 394 buffe
Mar 20 15:32:18 kaboom postgres[24396]: [22-1] 2024-03-20 15:32:18 UTC [@//:24396]: [23] LOG:  database system is
shut down
Mar 20 15:32:18 kaboom systemd[1]: postgres.service: Succeeded.
Mar 20 15:32:18 kaboom systemd[1]: Stopped Postgres 15 (TPA).
```

Swap PGDATA directories for version upgrade

Next, swap the PG15 and PG16 data directories:

```
sudo mv /opt/postgres/data
/opt/postgres/dataold
sudo mv /opt/postgres/datanew
/opt/postgres/data
```

> **Important**
>
> If something goes wrong at some point during the procedure, you may want to roll back/revert a node to the older major version. To do this, rename directories again so that the current data directory, `/opt/postgres/data`, becomes `/opt/postgres/datafailed` and the old data directory, `/opt/postgres/dataold`, becomes the current data directory:
>
> ```
> sudo mv /opt/postgres/data
> /opt/postgres/datafailed
> sudo mv /opt/postgres/dataold
> /opt/postgres/data
> ```
>
> This rolls back/reverts the node to the previous major version of Postgres.

Verify upgrade feasibility

The `bdr_pg_upgrade` tool has a `--check` option, which performs a dry run of some of the upgrade process. You can use this option to ensure the upgrade goes smoothly.

However, first, you need a directory for the files created by `bdr_pg_upgrade`. For this example, create an `/upgrade` directory in the `/home` directory. Then give ownership of the directory to the user postgres.

```
sudo mkdir /home/upgrade
sudo chown postgres:postgres /home/upgrade
```

Next, navigate to `/home/upgrade` and run:

```
sudo -u postgres /usr/bin/bdr_pg_upgrade
\
  --old-bindir /usr/lib/postgresql/15/bin/ \
  --new-bindir /usr/lib/postgresql/16/bin/ \
  --old-datadir /opt/postgres/dataold/ \
  --new-datadir /opt/postgres/data/
\
  --database bdrdb \
  --check
```

The following is the output:

```
 Performing BDR Postgres Checks
 ------------------------------
Collecting pre-upgrade new cluster control data             ok
Checking new cluster state is shutdown                      ok
Checking BDR versions                                       ok

Passed all bdr_pg_upgrade checks, now calling pg_upgrade

Performing Consistency Checks
 ------------------------------
Checking cluster versions                                   ok
Checking database user is the install user                  ok
Checking database connection settings                       ok
Checking for prepared transactions                          ok
Checking for system-defined composite types in user tables  ok
Checking for reg* data types in user tables                 ok
Checking for contrib/isn with bigint-passing mismatch       ok
Checking for presence of required libraries                 ok
Checking database user is the install user                  ok
Checking for prepared transactions                          ok
Checking for new cluster tablespace directories             ok

*Clusters are compatible
```

> **Note**
>
> If you didn't initialize Postgres 16 with checksums using the `--data-checksums` option but did initialize checksums with your Postgres 15 instance, an error tells you about the incompatibility:
>
> ```
> old cluster uses data checksums but the new one does
> not
> ```

Execute the Postgres major version upgrade

You're ready to run the upgrade. On the target node, run:

```
sudo -u postgres /usr/bin/bdr_pg_upgrade
\
  --old-bindir /usr/lib/postgresql/15/bin/ \
  --new-bindir /usr/lib/postgresql/16/bin/ \
  --old-datadir /opt/postgres/dataold/ \
  --new-datadir /opt/postgres/data/
\
  --database bdrdb
```

The following is the expected output:

```
Performing BDR Postgres Checks
```

```
------------------------------
Collecting pre-upgrade new cluster control data             ok
Checking new cluster state is shutdown                      ok
Checking BDR versions                                       ok
Starting old cluster (if shutdown)                          ok
Connecting to old cluster                                   ok
Checking if bdr schema exists                               ok
Turning DDL replication off                                 ok
Terminating connections to database                         ok
Disabling connections to database                           ok
Waiting for all slots to be flushed                         ok
Disconnecting from old cluster                              ok
Stopping old cluster                                        ok
Starting old cluster with BDR disabled                      ok
Connecting to old cluster                                   ok
Collecting replication origins                              ok
Collecting replication slots                                ok
Disconnecting from old cluster                              ok
Stopping old cluster                                        ok

Passed all bdr_pg_upgrade checks, now calling pg_upgrade

Performing Consistency Checks
------------------------------
Checking cluster versions                                   ok
Checking database user is the install user                  ok
Checking database connection settings                       ok
Checking for prepared transactions                          ok
Checking for system-defined composite types in user tables  ok
Checking for reg* data types in user tables                 ok
Checking for contrib/isn with bigint-passing mismatch       ok
Creating dump of global objects                             ok
Creating dump of database schemas                           ok
Checking for presence of required libraries                 ok
Checking database user is the install user                  ok
Checking for prepared transactions                          ok
Checking for new cluster tablespace directories             ok

If pg_upgrade fails after this point, you must re-initdb the
new cluster before continuing.

Performing Upgrade
------------------
Analyzing all rows in the new cluster                       ok
Freezing all rows in the new cluster                        ok
Deleting files from new pg_xact                             ok
Copying old pg_xact to new server                           ok
Setting oldest XID for new cluster                          ok
Setting next transaction ID and epoch for new cluster       ok
Deleting files from new pg_multixact/offsets                ok
Copying old pg_multixact/offsets to new server              ok
Deleting files from new pg_multixact/members                ok
Copying old pg_multixact/members to new server              ok
Setting next multixact ID and offset for new cluster        ok
Resetting WAL archives                                      ok
Setting frozenxid and minmxid counters in new cluster       ok
Restoring global objects in the new cluster                 ok
Restoring database schemas in the new cluster               ok
Copying user relation files                                 ok
Setting next OID for new cluster                            ok
Sync data directory to disk                                 ok
Creating script to delete old cluster                       ok
Checking for extension updates                              notice

Your installation contains extensions that should be updated
```

```
with the ALTER EXTENSION command.  The file
    update_extensions.sql
when executed by psql by the database superuser will update
these extensions.


Upgrade Complete
----------------
Optimizer statistics are not transferred by pg_upgrade.
Once you start the new server, consider running:
    /usr/pgsql-15/bin/vacuumdb --all --analyze-in-stages

Running this script will delete the old cluster's data files:
    ./delete_old_cluster.sh

pg_upgrade complete, performing BDR post-upgrade steps
------------------------------------------------------
Collecting old cluster control data                         ok
Collecting new cluster control data                         ok
Checking LSN of new cluster                                 ok
Starting new cluster with BDR disabled                      ok
Connecting to new cluster                                   ok
Creating replication origin (bdr_bdrdb_rb69_bdr2)           ok
Advancing replication origin (bdr_bdrdb_rb69_bdr2, 0/1F4... ok
Creating replication origin (bdr_bdrdb_rb69_bdr1)           ok
Advancing replication origin (bdr_bdrdb_rb69_bdr1, 0/1E8... ok
Creating replication slot (bdr_bdrdb_rb69_bdr1)             ok
Creating replication slot (bdr_bdrdb_rb69)                  ok
Creating replication slot (bdr_bdrdb_rb69_bdr2)             ok
Stopping new cluster
```

Update the Postgres service configuration

The Postgres service on the system is configured to start the old version of Postgres (PG15). You need to modify the `postgres.service` file to start the new version (PG16).

You can do this using `sed` to replace the old version number `15` with `16` throughout the file.

```
sudo sed -i -e  's/15/16/g'
/etc/systemd/system/postgres.service
```

After you've changed the version number, you can tell the systemd daemon to reload the configuration. On the target node, run:

```
sudo systemctl daemon-reload
```

Restart Postgres

Start the modified Postgres service:

```
sudo systemctl start
postgres
```

Validate the new Postgres version

Repeating the first step, check the version of Postgres to confirm that you upgraded kaboom correctly. While still on kaboom, run:

```
sudo -u postgres pgd show-
version
```

Use the output to confirm that kaboom is running the upgraded Postgres version:

```
 Node   BDR Version Postgres Version
----   ----------- ----------------
kaboom 5.4.0       16.2 (Debian 16.2-2EDB.buster)
kaftan 5.4.0       15.6 (Debian 15.6-2EDB.buster)
kaolin 5.4.0       15.6 (Debian 15.6-2EDB.buster)
```

Here kaboom has been upgraded to major version 16.

Clean up post-upgrade

As a best practice, run a vacuum over the database at this point. When the upgrade ran, you may have noticed the post-upgrade report included:

```
Once you start the new server, consider running:
    /usr/lib/postgresql/16/bin/vacuumdb --all --analyze-in-stages
```

You can run the vacuum now. On the target node, run:

```
sudo -u postgres /usr/lib/postgresql/16/bin/vacuumdb --all --analyze-in-
stages
```

If you're sure you don't need to revert this node, you can also clean up the old data directory folder `dataold`:

```
sudo rm -r
/opt/postgres/dataold
```

Upgrading the target node is now complete.

Next steps

After completing the upgrade on kaboom, run the same steps on kaolin and kaftan.

If you followed along with this example and kaftan is the write leader, to ensure availability, you mustperform a planned switchover to another node that was already upgraded before running the upgrade steps on kaftan.

Check Postgres versions across the cluster

After completing the upgrade on all nodes, while connected to one of the nodes, you can check your versions again:

```
sudo -u postgres pgd show-
version
```

The output will be similar to the following:

```
 Node   BDR Version Postgres Version
----   ----------- ----------------
kaboom 5.4.0       16.2 (Debian 16.2-2EDB.buster)
kaftan 5.4.0       16.2 (Debian 16.2-2EDB.buster)
kaolin 5.4.0       16.2 (Debian 16.2-2EDB.buster)
```

This output shows that all the nodes are successfully upgraded to the new Postgres version 16.

Reconcile with TPA

After all the nodes are upgraded, you still need to reconcile the upgraded version of Postgres with TPA so you can continue to use TPA to manage the cluster in the future.

To do this, return to the command line where your TPA cluster directory resides. In this worked example, the TPA cluster directory is `/home/ubuntu/democluster` on the instance where you originally deployed the cluster using TPA.

After navigating to your cluster directory, use a code editor to edit `config.yml` and change `cluster_vars:` from `postgres_version: '15'` to `postgres_version: '16'`.

Unless they were already added to your `.bashrc` or `.bash_profile`, ensure the TPA tools are accessible in your command line session by adding TPA's binary directory to your PATH:

```
export PATH=$PATH:/opt/EDB/TPA/bin
```

Finally, redeploy the cluster:

```
tpaexec deploy
democluster
```

This command applies the configuration changes to the cluster managed by TPA. If the deployment is successful, the reconciliation of the new version of Postgres with TPA and the upgrade procedure as a whole is complete.

20 Data migration to EDB Postgres Distributed

Moving data from one data source to another is a common task in the world of data management. This section provides information on how to migrate data to EDB Postgres Distributed from various data sources.

20.1 EDB*Loader and PGD

EDB*Loader is a high-speed data loading utility for EDB Postgres Advanced Server. It provides an interface compatible with Oracle databases, allowing you to load data into EDB Postgres Advanced Server. It's designed to load large volumes of data into EDB Postgres Advanced Server quickly and efficiently.

The EDB*Loader command line utility loads data from an input source into one or more tables using a subset of the parameters offered by Oracle SQL*Loader. The source can be a flat file, pipe, or other programs.

Use with PGD

As EDB*Loader is a utility for EDB Postgres Advanced Server, it's available for EDB Postgres Distributed when EDB Postgres Advanced Server is the database in use for PGD data nodes. PGD deployments can use EDB*Loader in the same way as it's used on EDB Postgres Advanced Server. See the EDB*Loader documentation for more information on how to use EDB*Loader with EDB Postgres Advanced Server.

Replication and EDB*Loader

As with EDB Postgres Advanced Server, EDB*Loader works with PGD in a replication environment. You cannot use the direct load path method because the direct path load method skips use of the WAL, upon which all replication relies. That means that only the node connected to by EDB*Loader gets the data that EDB*Loader is loading and no data replicates to the other nodes.

With PGD, you can make use of EDB*loader's direct load path method by running it independently on each node. You can perform this either on one node at a time or in parallel to all nodes, depending on the use case. When using the direct path load method on multiple nodes, it's important to ensure there are no other writes happening to the table concurrently as this can result in inconsistencies.

21 EDB Postgres Distributed Command Line Interface (PGD CLI)

The EDB Postgres Distributed Command Line Interface (PGD CLI) is a tool for managing your EDB Postgres Distributed cluster. It's the key tool for inspecting and managing cluster resources.

It allows you to run commands against EDB Postgres Distributed clusters to:

- Determine the health of the cluster, inspect the cluster's configuration, and manage the cluster's resources.
- Inspect and manage the cluster's nodes, groups, and proxies.
- Perform switchover operations on the write leaders of groups.

PGD CLI is installed automatically on systems in a TPA-deployed PGD cluster.

You can also install it manually on Linux and macOS systems that can connect to a PGD cluster, including:

- EDB BigAnimal distributed high-availability clusters.
- PGD clusters deployed using the EDB PGD for Kubernetes operator.
- Manually deployed PGD clusters.
- TPA-deployed PGD clusters.

21.1 Installing PGD CLI

You can install PGD CLI on any system that can connect to the PGD cluster. Linux and macOS are currently supported platforms to install PGD CLI on.

21.1.1 Installing PGD CLI on Linux

PGD CLI is available for most Linux distributions. You can install it from the EDB repositories, which you can access with your EDB account. PGD users and BigAnimal users, including those on a free trial, have an EDB account and access to PGD CLI.

Obtain your EDB subscription token

These repositories require a token to enable downloads from them. To obtain your token, log in to EDB Repos 2.0. If this is your first time visiting the EDB Repos 2.0 page, you must select **Request Access** to generate your token. Once a generated token is available, select the **Copy** icon to copy it to your clipboard, or select the eye icon to view it.

Set the EDB_SUBSCRIPTION_TOKEN environment variable

Once you have the token, execute the command shown for your operating system, substituting your token for `<your-token>`.

```
export EDB_SUBSCRIPTION_TOKEN=<your-token>
```

Then run the appropriate commands for your operating system.

Install on Debian or Ubuntu

```
curl -1sSLf "https://downloads.enterprisedb.com/$EDB_SUBSCRIPTION_TOKEN/postgres_distributed/setup.deb.sh" | sudo
-E bash
```

If this command returns an error like `curl: (22) The requested URL returned error: 404`, check that you entered the correct token.

When the command is successful, you'll see output like this:

```
Executing the  setup script for the 'enterprisedb/postgres_distributed' repository ...
...
```

You can now install the PGD CLI package using the command:

```
sudo apt-get install edb-pgd5-cli
```

Install on RHEL, Rocky, AlmaLinux, or Oracle Linux

```
curl -1sSLf "https://downloads.enterprisedb.com/$EDB_SUBSCRIPTION_TOKEN/postgres_distributed/setup.rpm.sh" | sudo
-E bash
```

If this command returns an error like `curl: (22) The requested URL returned error: 404`, check that you entered the correct token.

When the command is successful, you'll see output like this:

```
Executing the  setup script for the 'enterprisedb/postgres_distributed' repository ...
...
```

You can now install the PGD CLI package using the command:

```
sudo yum install edb-pgd5-cli
```

Next: Using PGD CLI

21.1.2 Installing PGD CLI on macOS

PGD CLI is available for macOS as a Homebrew formula. To install it, run the following commands:

```
brew tap enterprisedb/tap
brew install pgd-cli
```

To verify the installation, run:

```
pgd --version
```

Next: Using PGD CLI

21.1.3 Installing PGD CLI with TPA

By default, Trusted Postgres Architect installs and configures PGD CLI on each PGD node.

If you want to install PGD CLI on any non-PGD instance in the cluster, attach the pgdcli role to that instance in Trusted Postgres Architect's configuration file before deploying.

See Trusted Postgres Architect for more information.

21.2 Using PGD CLI

What is the PGD CLI?

The PGD CLI is a convenient way to connect to and manage your PGD cluster. To use it, you need a user with PGD superuser privileges or equivalent. The PGD user with superuser privileges is the bdr_superuser role. An example of an equivalent user is edb_admin on an EDB BigAnimal distributed high-availability cluster.

> **Setting passwords**
>
> PGD CLI doesn't interactively prompt for your password. You must pass your password using one of the following methods:
>
> - Adding an entry to your `.pgpass` password file, which includes the host, port, database name, user name, and password.
> - Setting the password in the `PGPASSWORD` environment variable.
> - Including the password in the connection string.
>
> We recommend the first option, as the other options don't scale well with multiple databases, or they compromise password confidentiality.

Running the PGD CLI

Once you have installed pgd-cli, run the `pgd` command to access the PGD command line interface. The `pgd` command needs details about the host, port, and database to connect to, along with your username and password.

Passing a database connection string

Use the `--dsn` flag to pass a database connection string to the `pgd` command. When you pass the connection string with the `--dsn` flag, you don't need a configuration file. The flag takes precedence even if a configuration file is present. For example:

```
pgd show-nodes --dsn "host=bdr-a1 port=5432 dbname=bdrdb user=enterprisedb"
```

See pgd in the command reference for a description of the command options.

Specifying a configuration file

If a `pgd-cli-config.yml` file is in `/etc/edb/pgd-cli` or `$HOME/.edb/pgd-cli`, `pgd` uses it. You can override this behavior using the optional `-f` or `--config-file` flag. For example:

```
pgd show-nodes -f /opt/my-
config.yml
Node              Node ID    Group          Type    Current State Target State Status Seq
ID
----              -------    -----          ----    ------------- ------------ ------ ------
p-vjljj303dk-a-1 2573417965 p-vjljj303dk-a data    ACTIVE        ACTIVE       Up
1
p-vjljj303dk-a-2 126163807  p-vjljj303dk-a data    ACTIVE        ACTIVE       Up
2
p-vjljj303dk-a-3 3521351376 p-vjljj303dk-a witness ACTIVE        ACTIVE       Up
3
```

Specifying the output format

Use the `-o` or `--output` flag to change the default output format to JSON or YAML. For example:

```
pgd show-nodes -o json
[

{
        "node_id":
2573417965,
        "node_name": "p-vjljj303dk-a-1",
        "node_group_id":
4169125197,
        "node_group_name": "p-vjljj303dk-a",
        "node_kind_name": "data",
        "current_state": "ACTIVE",
        "target_state": "ACTIVE",
        "status": "Up",
        "node_seq_id": 1,
        "node_local_dbname": "bdrdb",
        "interface_connstr": "host=p-vjljj303dk-a-1-node.vmk31wilqpjeopka.biganimal.io user=streaming_replica
sslmode=verify-full port=5432 sslkey=/controller/certificates/streaming_replica.key
sslcert=/controller/certificates/streaming_replica.crt sslrootcert=/controller/certificates/server-ca.crt
application_name=p-vjljj303dk-a-1 dbname=bdrdb",
        "route_priority": -1,
        "route_fence":
false,
        "route_writes": true,
        "route_reads": true,
        "route_dsn": "host=p-vjljj303dk-a-1-node.vmk31wilqpjeopka.biganimal.io user=streaming_replica
sslmode=verify-full port=5432 sslkey=/controller/certificates/streaming_replica.key
sslcert=/controller/certificates/streaming_replica.crt sslrootcert=/controller/certificates/server-ca.crt
application_name=p-vjljj303dk-a-1 dbname=bdrdb"
    },
...
]
```

The PGD CLI supports the following output formats.

Setting	Format	Considerations
none	Tabular	Default format. This setting presents the data in tabular form.
`json`	JSON	Presents the raw data with no formatting. For some commands, the JSON output might show more data than the tabular output, such as extra fields and more detailed messages.
`yaml`	YAML	Similar to the JSON output but as YAML and with the fields ordered alphabetically. Experimental and might not be fully supported in future versions.

Accessing the command line help

To list the supported commands, enter:

```
pgd help
```

For help with a specific command and its parameters, enter `pgd help <command_name>`. For example:

```
pgd help show-nodes
```

> **Avoiding stale data**
>
> The PGD CLI can return stale data on the state of the cluster if it's still connecting to nodes previously parted from the cluster. Edit the `pgd-cli-config.yml` file, or change your `--dsn` settings to ensure you are connecting to active nodes in the cluster.

21.3 Configuring PGD CLI

PGD CLI can be installed on any system that can connect to the PGD cluster. To use PGD CLI, you need a user with PGD superuser privileges or equivalent. The PGD user with superuser privileges is the bdr_superuser role. An example of an equivalent user is edb_admin on a BigAnimal distributed high-availability cluster.

PGD CLI and database connection strings

You might not need a database connection string. For example, when Trusted Postgres Architect installs the PGD CLI on a system, it also configures the connection to the PGD cluster, which means that the PGD CLI can connect to the cluster when run.

If you're installing PGD CLI manually, you must give PGD CLI a database connection string so it knows which PGD cluster to connect to.

> **Setting passwords**
>
> PGD CLI doesn't interactively prompt for your password. You must pass your password using one of the following methods:
>
> - Adding an entry to your `.pgpass` password file, which includes the host, port, database name, user name, and password.
> - Setting the password in the `PGPASSWORD` environment variable.
> - Including the password in the connection string.
>
> We recommend the first option, as the other options don't scale well with multiple databases, or they compromise password confidentiality.

If you don't know the database connection strings for your PGD-powered deployment, see discovering connection strings, which helps you to find the right connection strings for your cluster.

Once you have that information, you can continue.

Configuring the database to connect to

PGD CLI takes its database connection information from either the PGD CLI configuration file or the command line.

Using database connection strings in the command line

You can pass the connection string directly to `pgd` using the `--dsn` option. For details, see the sample use case. For example:

```
pgd --dsn "host=bdr-a1 port=5432 user=enterprisedb" show-version
```

Using a configuration file

Use the `pgd-cli-config.yml` configuration file to specify the database connection string for your cluster. The configuration file must contain the database connection string for at least one PGD node in the cluster. The cluster name is optional and isn't validated.

For example:

```
cluster:
  name: cluster-
name
  endpoints:
  - "host=bdr-a1 port=5432 dbname=bdrdb user=enterprisedb"
  - "host=bdr-b1 port=5432 dbname=bdrdb user=enterprisedb"
  - "host=bdr-c1 port=5432 dbname=bdrdb user=enterprisedb"
```

By default, `pgd-cli-config.yml` is located in the `/etc/edb/pgd-cli` directory. The PGD CLI searches for `pgd-cli-config.yml` in the following locations. Precedence order is high to low.

1. `/etc/edb/pgd-cli` (default)
2. `$HOME/.edb/pgd-cli`

If your configuration file isn't in either of these directories, you can use the optional `-f` or `--config-file` flag on a `pgd` command to set the file to read as configuration. See the sample use case.

21.4 Discovering connection strings

You can install PGD CLI on any system that can connect to the PGD cluster. To use PGD CLI, you need a user with PGD superuser privileges or equivalent. The PGD user with superuser privileges is the bdr_superuser role. An example of an equivalent user is edb_admin on an EDB BigAnimal distributed high-availability cluster.

PGD CLI and database connection strings

You might not need a database connection string. For example, when Trusted Postgres Architect installs the PGD CLI on a system, it also configures the connection to the PGD cluster. This means that PGD CLI can connect to the cluster when run.

Getting your database connection string

Because of the range of different configurations that PGD supports, every deployment method has a different way of deriving a connection string for it. Generally, you can obtain the required information from the configuration of your deployment. You can then assemble that information into connection strings.

For a TPA-deployed PGD cluster

Because TPA is so flexible, you have to derive your connection string from your cluster configuration file (`config.yml`).

- You need the name or IP address of a host with the role pgd-proxy listed for it. This host has a proxy you can connect to. Usually the proxy listens on port 6432. (Check the setting for `default_pgd_proxy_options` and `listen_port` in the config to confirm.)
- The default database name is `bdrdb`. (Check the setting `bdr_database` in the config to confirm.)
- The default PGD superuser is enterprisedb for EDB Postgres Advanced Server and postgres for PostgreSQL and EDB Postgres Extended Server.

You can then assemble a connection string based on that information:

```
"host=<hostnameOrIPAddress> port=<portnumber> dbname=<databasename> user=<username> sslmode=require"
```

To illustrate this, here are some excerpts of a `config.yml` file for a cluster:

```
 ...
cluster_vars:

...
  bdr_database: bdrdb

...

default_pgd_proxy_options:
    listen_port: 6432

...

instances:
- Name:
kaboom
  backup: kapok
  location:
dc1
  node:
1
  role:
  -
bdr
  - pgd-
proxy
  networks:
  - ipv4_address:
192.168.100.2
    name:
tpanet
...
```

The connection string for this cluster is:

```
"host=192.168.100.2 port=6432 dbname=bdrdb user=enterprisedb sslmode=require"
```

> **Host name versus IP address**
>
> The example uses the IP address because the configuration is from a Docker TPA install with no name resolution available. Generally, you can use the host name as configured.

For an EDB BigAnimal distributed high-availability cluster

1. Log in to the BigAnimal clusters view.
2. In the filter, set **Cluster Type** to **Distributed High Availability** to show only clusters that work with PGD CLI.
3. Select your cluster.
4. In the view of your cluster, select the **Connect** tab.
5. Copy the read/write URI from the connection info. This is your connection string.

For a cluster deployed with EDB PGD for Kubernetes

As with TPA, EDB PGD for Kubernetes is very flexible, and there are multiple ways to obtain a connection string. It depends, in large part, on the configuration of the deployment's services:

- If you use the Node Service Template, direct connectivity to each node and proxy service is available.
- If you use the Group Service Template, there's a gateway service to each group.
- If you use the Proxy Service Template, a single proxy provides an entry point to the cluster for all applications.

Consult your configuration file to determine this information.

Establish a host name or IP address, port, database name, and username. The default database name is `bdrdb`. The default username is enterprisedb for EDB Postgres Advanced Server and postgres for PostgreSQL and EDB Postgres Extended Server.

You can then assemble a connection string based on that information:

```
"host=<hostnameOrIPAddress> port=<portnumber> dbname=<databasename> user=<username>"
```

If the deployment's configuration requires it, add `sslmode=<sslmode>`.

21.5 Command reference

The command name for the PGD command line interface is `pgd`.

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.

Global Options

All commands accept the following global options:

Short	Long	Description
	--dsn	Database connection string For example "host=bdr-a1 port=5432 dbname=bdrdb user=postgres"
-f	--config-file	Name/Path to config file. This is ignored if --dsn flag is present Default "/etc/edb/pgd-cli/pgd-cli-config.yml"
-h	--help	Help for pgd - will show specific help for any command used
-L	--log-level	Logging level: debug, info, warn, error (default "error")
-o	--output	Output format: json, yaml

See also

- check-health
- create-proxy
- delete-proxy
- set-group-options
- set-node-options
- set-proxy-options
- show-clockskew
- show-events
- show-groups
- show-nodes
- show-proxies
- show-raft
- show-replslots
- show-subscriptions
- show-version
- switchover
- verify-cluster
- verify-settings

21.5.1 check-health

Checks the health of the EDB Postgres Distributed cluster.

Synopsis

Performs various checks such as if all nodes are accessible and all replication slots are working.

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]
```

Options

No specific command options. See global options for global options.

Examples

Checking health with a node down

In this example, we have a 3 node cluster, bdr-a1 and bdr-c1 are up, bdr-b1 is down.

```
$ pgd check-
health
```

output

```
 Check      Status   Message
 -----      ------   -------
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 at least 1 node that is not accessible
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
```

Checking health with clock skew

In this example there is a 3 node cluster with all nodes up but the system clocks are not in sync.

```
$ pgd check-
health
```

output

```
 Check      Status  Message
 -----      ------  -------
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
```

Checking health with all nodes working correctly

In this example, there is a 3 node cluster with all nodes are up and all checks are Ok.

```
$ pgd check-
health
```

output

```
 Check      Status Message
-----      ------ -------
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
```

21.5.2 create-proxy

Creates proxy in the EDB Postgres Distributed cluster.

Synopsis

Creates proxy in the EDB Postgres Distributed cluster and attaches it to the given group. The proxy name must be unique across the cluster and match with the name given in the corresponding proxy config file.

Use the proxy mode to route connections to Write Leader (default), Read Nodes (read-only), or both (any). Proxy listens on 'listen_port' for Write Leader connections while on 'read_listen_port' for Read Nodes connections.

```
pgd create-proxy [flags]
```

Options

Flag	Description
`--group-name`	Group name
`--proxy-mode`	Proxy mode (default, read-only, any); proxy will route connections to - default - Write Leader read-only - Read Nodes any - both Write Leader and Read Nodes (default "default")
`--proxy-name`	Proxy name

See global options for global options.

Examples

Attaching in default mode.

In this example, we attach a new proxy called proxy-a1 to group group_a, with 'default' mode.

```
$ pgd create-proxy --proxy-name proxy-a1 --group-name
group_a
```

output

```
proxy created successfully
```

Attaching in any mode.

In this example, we attach anew proxy called proxy-b1 to group group_b, with 'any' mode.

```
$ pgd create-proxy --proxy-name proxy-b1 --group-name group_b --proxy-mode
any
```

output

```
proxy created successfully
```

21.5.3 delete-proxy

Deletes a proxy from the EDB Postgres Distributed cluster.

Synopsis

Deletes a proxy from the EDB Postgres Distributed cluster.

```
pgd delete-proxy [flags]
```

Options

Flag	Description
`--proxy-name`	proxy name

See global options for global options.

Examples

Deleting a proxy

```
$ pgd delete-proxy --proxy-name proxy-
a1
```

output

```
proxy deleted successfully
```

21.5.4 set-group-options

Sets group options such as `enable_raft`, `enable_proxy_routing`, and `location`.

Synopsis

You can set the following group options with this command:

- `enable_raft`
- `enable_proxy_routing`
- `location`
- `route_writer_max_lag`
- `route_reader_max_lag`

Both `enable_raft` and `enable_proxy_routing` must be true if proxy is attached to the group.

Use `pgd show-groups -o json` to view option values for each group.

```
pgd set-group-options [flags]
```

Options

Flag	Description
`--group-name`	group name
`--option`	option in name=value format

See global options for global options.

Examples

Setting group options with multiple options

In this example, we use comma separated multiple options. Spaces are not allowed in the option values.

```
$ pgd set-group-options --group-name bdrgroup --option
enable_proxy_routing=true,route_writer_max_lag=1000000
```

output

```
group options updated successfully
```

Setting group options with multiple option flags

In this example, we use multiple option flags. Spaces are not allowed in the option values.

```
$ pgd set-group-options --group-name bdrgroup --option enable_proxy_routing=true --option
route_writer_max_lag=1000000
```

output

```
group options updated successfully
```

Setting group options with double quotes

In this example, we use double quotes around options if the option value has spaces or special characters.

```
$ pgd set-group-options --group-name bdrgroup --option "location = mumbai" --option "route_writer_max_lag =
1000000"
```

output

```
group options updated successfully
```

21.5.5 set-node-options

Sets node options such as `route_fence`, `route_priority`, and `route_writes`.

Synopsis

You can set the following node options with this command:

- `route_dsn`
- `route_fence`
- `route_priority`
- `route_writes`
- `route_reads`

Use `pgd show-nodes -o json` to view option values for each node.

```
pgd set-node-options [flags]
```

Options

Flag	Description
`--node-name`	node name
`--option`	option in name=value format

See global options for global options.

Examples

Setting node options with multiple options

In this example, we use comma separated multiple options. Spaces are not allowed in the option values.

```
$ pgd set-node-options --node-name bdr-a1 --option
route_priority=100,route_fence=true
```

output

```
node options updated successfully
```

Setting node options with multiple option flags

In this example, we use multiple option flags. Spaces are not allowed in the option values.

```
$ pgd set-node-options --node-name bdr-a1 --option route_priority=100 --option
route_fence=true
```

output

```
node options updated successfully
```

Setting node options with double quotes

In this example, we use double quotes around options if the option value has spaces or special characters.

```
$ pgd set-node-options --node-name bdr-a1 --option "route_priority = 100" --option "route_fence =
true"
```

output

```
node options updated successfully
```

21.5.6 set-proxy-options

Sets proxy options such as `listen_address`, `listen_port`, and `max_client_conn`.

Synopsis

You can set the following proxy options with this command:

- `listen_address`
- `listen_port`
- `max_client_conn`
- `max_server_conn`
- `server_conn_keepalive`
- `server_conn_timeout`
- `consensus_grace_period`
- `read_listen_address`
- `read_listen_port`
- `read_max_client_conn`
- `read_max_server_conn`
- `read_server_conn_keepalive`
- `read_server_conn_timeout`
- `read_consensus_grace_period`

After updating any of these options, restart proxy.

Set `listen_port` to non-zero value to route traffic to the Write Leader and set `read_listen_port` to non-zero value to route traffic to Read nodes. Setting it to zero will disable the respective routing.

Use `pgd show-proxies -o json` to view option values for each proxy.

```
pgd set-proxy-options [flags]
```

Options

Flag	Description
`--proxy-name`	proxy name
`--option`	option in name=value format

See global options for global options.

Examples

Setting proxy options with multiple options

In this example, we use comma separated multiple options. Spaces are not allowed in the option values.

```
$ pgd set-proxy-options --proxy-name proxy-a1 --option
listen_address=0.0.0.0,listen_port=6432
```

output

```
proxy options updated successfully, please restart proxy service
```

Setting proxy options with multiple option flags

In this example, we use multiple option flags. Spaces are not allowed in the option values.

```
$ pgd set-proxy-options --proxy-name proxy-a1 --option listen_address=0.0.0.0 --option
listen_port=0
```

output

```
proxy options updated successfully, please restart proxy service
```

Setting proxy options with double quotes

In this example, we use double quotes around options if the option value has spaces or special characters.

```
$ pgd set-proxy-options --proxy-name proxy-a1 --option "listen_address = 0.0.0.0" --option
"consensus_grace_period=1h 30m 5s"
```

output

```
proxy options updated successfully, please restart proxy service
```

21.5.7 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]
```

Options

No specific command options. See global options for global options.

Examples

Show clock skew with a node down

In this example, there is a 3 node cluster, bdr-a1 and bdr-c1 are up, bdr-b1 is down.

```
$ pgd show-
clockskew
```

output

```
 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
```

Show clock skew with all nodes working correctly

In this example, there is a 3 node cluster with all nodes are up and all clocks are in sync.

```
$ pgd show-
clockskew
```

output

```
 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
```

21.5.8 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]
```

Node States

State	Description
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 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 complete; 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. STANDBY indicates that the node is in a read-only state.

Options

Flag	Description
`-n, --lines`	show top n lines

See global options for global options.

Examples

Showing top 10 events

In this example, we show top 10 events on a three node cluster.

```
$ pgd show-events --lines
10
```

output

```
Time                          Observer Node Subject Node Source    Type   Subtype      Message
----                          ------------- ------------ ------    ----   -------      -------
2023-03-23 05:38:25.243257+00 witness-a1    witness-a1   consensus RAFT   STATE_CHANGE RAFT_LEADER
2023-03-23 05:38:25.23815+00  witness-a1    witness-a1   consensus RAFT   STATE_CHANGE RAFT_CANDIDATE
2023-03-23 05:38:21.197974+00 bdr-a1        bdr-a1       consensus RAFT   STATE_CHANGE RAFT_FOLLOWER
2023-03-23 05:38:21.197107+00 witness-a1    witness-a1   consensus RAFT   STATE_CHANGE RAFT_FOLLOWER
2023-03-23 05:38:21.169781+00 bdr-a2        bdr-a2       consensus RAFT   STATE_CHANGE RAFT_FOLLOWER
2023-03-23 05:38:17.949669+00 witness-a1    bdr-a1       consensus NODE   STATE_CHANGE ACTIVE
2023-03-23 05:38:17.949544+00 bdr-a1        bdr-a1       consensus NODE   STATE_CHANGE ACTIVE
2023-03-23 05:38:17.946857+00 bdr-a2        bdr-a1       consensus NODE   STATE_CHANGE ACTIVE
2023-03-23 05:38:17.91628+00  bdr-a1        bdr-a2       receiver  WORKER ERROR        pglogical worker received
fast finish request, exiting
2023-03-23 05:38:17.915236+00 witness-a1    bdr-a1       consensus NODE   STATE_CHANGE PROMOTING
```

21.5.9 show-groups

Shows all groups in the EDB Postgres Distributed cluster and their summary.

Synopsis

Shows all groups in the EDB Postgres Distributed cluster and their summary, including type, parent group, location, Raft and Routing status, Raft leader, and write leader.

In some cases when Raft isn't working properly or the group Raft leader isn't present, this command might show stale or incorrect write leader for that group.

```
pgd show-groups [flags]
```

Options

No specific command options. See global options for global options.

Examples

Show all groups in the cluster

In this example, there is a 4 group cluster, 3 data groups and one subscriber-only group. `bdrgroup` is the global group. `group_a` , `group_b` and `group_c` are data groups. `group_so` is the subscriber-only group.

Note:

1. For write leader election both Raft and Routing options should be true for that group.
2. Raft is always true for global group.

```
$ pgd show-
groups
```

output

```
Group    Group ID   Type            Parent Group Location Raft  Routing Raft Leader Write Leader
-----    --------   ----            ------------ -------- ----  ------- ----------- ------------
bdrgroup 1360502012 global                       world    true  false   bdr-a2
group_a  3618712053 data            bdrgroup     a        true  true    bdr-a2      bdr-a1
group_b  402614658  data            bdrgroup     b        true  true    bdr-b1      bdr-b1
group_c  2808307099 data            bdrgroup     c        false false
group_so 2123208041 subscriber-only bdrgroup     c        false false
```

21.5.10 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

State	Description
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 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 complete; 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. STANDBY, in the Current State or Target State columns, indicates that the node is or will be in a read-only state.

```
pgd show-nodes [flags]
```

Options

No specific command options. See global options for global options.

Examples

Show all nodes in the cluster with a node down

In this example, there is a multi-node cluster with a data node down.

```
$ pgd show-
nodes
```

output

```
Node      Node ID    Group   Type    Current State Target State Status      Seq ID
----      -------    -----   ----    ------------- ------------ ------      ------
bdr-a1    3136956818 group_a data    ACTIVE        ACTIVE       Up          1
bdr-a2    2133699692 group_a data    ACTIVE        ACTIVE       Unreachable 2
witness-a 3889635963 group_a witness ACTIVE        ACTIVE       Up          3
```

Show all nodes in the cluster with different node types

In this example, there is a multi-node cluster with logical standby, witness and subscriber-only nodes. Note that, unlike logical standby nodes, the subscriber-only nodes are fully joined node to the cluster.

```
$ pgd show-
nodes
```

output

```
Node               Node ID    Group    Type            Current State Target State Status Seq ID
----               -------    -----    ----            ------------- ------------ ------ ------
bdr-a1             3136956818 group_a  data            ACTIVE        ACTIVE       Up     6
bdr-a2             2133699692 group_a  data            ACTIVE        ACTIVE       Up     3
logical-standby-a1 1140256918 group_a  standby         STANDBY       STANDBY      Up     9
witness-a          3889635963 group_a  witness         ACTIVE        ACTIVE       Up     7
bdr-b1             2380210996 group_b  data            ACTIVE        ACTIVE       Up     1
bdr-b2             2244996162 group_b  data            ACTIVE        ACTIVE       Up     2
logical-standby-b1 3541792022 group_b  standby         STANDBY       STANDBY      Up     10
witness-b          661050297  group_b  witness         ACTIVE        ACTIVE       Up     5
witness-c          1954444188 group_c  witness         ACTIVE        ACTIVE       Up     4
subscriber-only-c1 2448841809 group_so subscriber-only ACTIVE        ACTIVE       Up     8
```

21.5.11 show-proxies

Shows all proxies in the EDB Postgres Distributed cluster and their summary.

Synopsis

Shows all proxies in the EDB Postgres Distributed cluster and their summary.

We recommend giving all the proxies attached to the same group the same proxy option values.

```
pgd show-proxies [flags]
```

Options

No specific command options. See global options for global options.

Examples

Show all proxies in the cluster

In this example, there is a multi-group cluster, with 2 proxies attached to each data group.

```
$ pgd show-
proxies
```

output

```
 Proxy    Group   Listen Addrs Listen Port Read Listen Addrs Read Listen Port
------    -----   ------------ ----------- ----------------- ----------------
proxy-a1 group_a [0.0.0.0]    6432        [0.0.0.0]         6433
proxy-a2 group_a [0.0.0.0]    6432        [0.0.0.0]         6433
proxy-b1 group_b [0.0.0.0]    6432        [0.0.0.0]         6433
proxy-b2 group_b [0.0.0.0]    6432        [0.0.0.0]         6433
```

21.5.12 show-raft

Shows BDR Raft (consensus protocol) details.

Synopsis

Shows BDR Raft (consensus protocol) details such as Raft instance id, Raft state (leader, follower), and Raft term. If Raft is enabled at subgroup level, then that subgroup's Raft instance is also shown.

In some cases, such as network partition, output might vary based on the node to which the CLI is connected.

```
pgd show-raft [flags]
```

Options

No specific command options. See global options for global options.

Examples

Show Raft details

In this example, there is a multi-group cluster with subgroup Raft and with witness, logical standby, subscriber-only nodes. Note that logical standby and subscriber-only nodes don't have Raft voting rights, unlike data or witness nodes.

```
$ pgd show-raft
```

output

Instance	Group	Node	Raft State	Raft Term	Commit Index	Nodes	Voting Nodes	Protocol Version
1	bdrgroup	bdr-b1	RAFT_LEADER	0	383	10	7	5000
1	bdrgroup	bdr-a1	RAFT_FOLLOWER	0	383	10	7	5000
1	bdrgroup	bdr-a2	RAFT_FOLLOWER	0	383	10	7	5000
1	bdrgroup	bdr-b2	RAFT_FOLLOWER	0	383	10	7	5000
1	bdrgroup	logical-standby-a1	RAFT_FOLLOWER	0	383	10	7	5000
1	bdrgroup	logical-standby-b1	RAFT_FOLLOWER	0	383	10	7	5000
1	bdrgroup	subscriber-only-c1	RAFT_FOLLOWER	0	383	10	7	5000
1	bdrgroup	witness-a	RAFT_FOLLOWER	0	383	10	7	5000
1	bdrgroup	witness-b	RAFT_FOLLOWER	0	383	10	7	5000
1	bdrgroup	witness-c	RAFT_FOLLOWER	0	383	10	7	5000
2	group_a	witness-a	RAFT_LEADER	1	2	4	3	0
2	group_a	bdr-a1	RAFT_FOLLOWER	1	2	4	3	0
2	group_a	bdr-a2	RAFT_FOLLOWER	1	2	4	3	0
2	group_a	logical-standby-a1	RAFT_FOLLOWER	1	2	4	3	0
3	group_b	witness-b	RAFT_LEADER	1	2	4	3	0
3	group_b	bdr-b1	RAFT_FOLLOWER	1	2	4	3	0
3	group_b	bdr-b2	RAFT_FOLLOWER	1	2	4	3	0
3	group_b	logical-standby-b1	RAFT_FOLLOWER	1	2	4	3	0

21.5.13 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 maxOf(WriteLag, FlushLag, ReplayLag, SentLag).

```
pgd show-replslots [flags]
```

Options

Flag	Description
-v, --verbose	verbose output

See global options for global options.

Examples

Show replication slots with a node down

In this example, there is a 3 node cluster, bdr-a1 and bdr-c1 are up, bdr-b1 is down.

```
$ pgd show-
replslots
```

output

```
Node   bdr-a1 bdr-b1  bdr-c1
----   ------ ------  ------
bdr-a1 *      !(6.6G) *
bdr-b1 x      *       x
bdr-c1 *      !(6.9G) *
```

Or in Verbose mode:

```
$ pgd show-replslots --
verbose
```

```
output
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
```

Show replication slots with a recently restarted node

In this example, there is a 3 node cluster, bdr-b1 was down and it has just been restarted.

```
$ pgd show-
replslots
```

```
output
Node   bdr-a1 bdr-b1  bdr-c1
----   ------ ------  ------
bdr-a1 *      !(6.9G) *
bdr-b1 *      *       *
bdr-c1 *      !(5.8G) *
```

Or in Verbose mode:

```
$ pgd show-replslots --
verbose
```

```
output
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
```

Show replication slots with all nodes working correctly

In this example, there is a 3 node cluster with all nodes are up and in 'streaming' state.

```
$ pgd show-
replslots
```

output

```
 Node   bdr-a1 bdr-b1 bdr-c1
----   ------ ------ ------
bdr-a1 *      *      *
bdr-b1 *      *      *
bdr-c1 *      *      *
```

Or in Verbose mode:

```
$ pgd show-replslots --
verbose
```

output

```
 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
```

Show replication slots in a multi-node cluster

In this example, there is a 4 node cluster, with a witness node, a subscriber-only node, and two logical standbys. bdr-a1 and bdr-b1 are up and in 'streaming' state. bdr-a1 is replicating to logical-standby-a1 and bdr-b1 is replicating to logical-standby-b1. bdr-a1 is also replicating to subscriber-only-c1.

Note:

1. Data for a logical standby is only sent by one source node. No other nodes receive replication changes from the logical standby.
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
```

output

```
 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        *      *      -                  -                  *                  *
```

21.5.14 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]
```

Options

No specific command options. See global options for global options.

Examples

Show subscriptions with a node down

In this example, there is a 3 node cluster, bdr-a1 and bdr-c1 are up, bdr-b1 is down.

```
$ pgd show-
subscriptions
```

output

```
 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
```

Show subscriptions with a recently restarted node

In this example, there is a 3 node cluster, bdr-b1 was down and it has just been restarted.

```
$ pgd show-
subscriptions
```

output

```
 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
```

Show subscriptions with all nodes working correctly

In this example, there is a 3 node cluster, all nodes are up and in 'streaming' state.

```
$ pgd show-
subscriptions
```

output

```
 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
```

Show subscriptions in a multi-node cluster

In this example, there is a 4 node cluster. bdr-a1 and bdr-b1 are the origin nodes for logical-standby-a1 and logical-standby-b1 respectively. bdr-a1 and bdr-b1 are the origin nodes for subscriber-only-c1. bdr-a1 and bdr-b1 are the origin nodes for witness-c1.

Note: Logical standby and subscriber-only nodes receive changes but do not send changes made locally to other nodes

```
$ pgd show-
subscriptions
```

output

```
 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
```

21.5.15 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]
```

Options

No specific command options. See global options for global options.

Examples

Show version with a node down

In this example, there is a 3 node cluster, bdr-a1 and bdr-c1 are up, bdr-b1 is down.

```
$ pgd show-
version
```

output

```
 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
```

Show version with all nodes up

In this example, there is a 3 node cluster, all nodes are up.

```
$ pgd show-
version
```

output

```
 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)
```

21.5.16 switchover

Switches over to new write leader.

Synopsis

Switches over to new write leader. Use switchover method `fast` for immediate switchover. Use `strict` to wait until lag is less than `route_writer_max_lag` on the given target node.

If switchover fails due to timeout or any other issue, BDR might elect a write leader that's different from the given target node.

```
pgd switchover [flags]
```

Options

Flag	Value	Description
`--group-name`	string	Group name
`--method`	string	Switchover method (strict, fast) strict - waits until lag on given node is less than route_writer_max_lag fast - immediate switchover, route_writer_max_lag is ignored (default "strict")
`--node-name`	string	Node name
`--timeout`	interval	Timeout period when switchover method is strict (default 10s)

See global options for global options.

Examples

Using defaults

Running the command with only required arguments. The default method is `strict` and default timeout is `10s`.

```
$ pgd switchover --group-name group_a --node-name bdr-
a1
```

output

```
switchover is complete
```

Using optional arguments

Running the command with optional arguments.

```
$ pgd switchover --group-name group_a --node-name bdr-a1 --method strict --timeout
15s
```

output

```
switchover is complete
```

Immediate switchover

Running the command with `fast` method for immediate switchover.

```
$ pgd switchover --group-name group_a --node-name bdr-a1 --method
fast
```

output

```
switchover is complete
```

21.5.17 verify-cluster

Verifies whether the cluster follows the rules as per the AlwaysOn architecture.

Synopsis

Verifies whether the cluster follows the rules as per the AlwaysOn architecture.

```
pgd verify-cluster [flags]
```

Options

No specific command options. See global options for global options.

Examples

Verifying a cluster with an unrecommended architecture

In this example, we verify the cluster with an unrecommended architecture.

```
$ pgd verify-
cluster
```

output

```
Check                                                                  Status   Groups
-----                                                                  ------   ------
There is always at least 1 Global Group and 1 Data Group               Ok
There are at least 2 data nodes in a Data Group (except for the witness-only group) Critical group_b
There is at most 1 witness node in a Data Group                        Warning  group_a
Witness-only group does not have any child groups                      Ok
There is at max 1 witness-only group iff there is even number of local Data Groups  Warning  bdrgroup
There are at least 2 proxies configured per Data Group if routing is enabled        Warning  group_a, group_b
```

Verifying a cluster with recommended architecture

In this example, we verify the cluster with a recommended architecture.

```
$ pgd verify-
cluster
```

output

```
Check                                                                  Status Groups
-----                                                                  ------ ------
There is always at least 1 Global Group and 1 Data Group               Ok
There are at least 2 data nodes in a Data Group (except for the witness-only group) Ok
There is at most 1 witness node in a Data Group                        Ok
Witness-only group does not have any child groups                      Ok
There is at max 1 witness-only group iff there is even number of local Data Groups  Ok
There are at least 2 proxies configured per Data Group if routing is enabled        Ok
```

21.5.18 verify-settings

Verifies the EDB Postgres Distributed cluster settings.

Synopsis

Verifies the EDB Postgres Distributed cluster settings.

```
pgd verify-settings [flags]
```

Options

No specific command options. See global options for global options.

Examples

Verifying the cluster settings

```
$ pgd verify-settings
```

output

```
 # bdr.accept_connections
Node               Status   Pending Restart Value Message
----               ------   --------------- ----- -------
bdr-a1             Critical false           off   must be set to on
bdr-a2             Ok       false           on
bdr-b1             Ok       false           on
bdr-b2             Ok       false           on
logical-standby-a1 Ok       false           on
logical-standby-b1 Ok       false           on
subscriber-only-c1 Ok       false           on
witness-a          Ok       false           on
witness-b          Ok       false           on
witness-c          Ok       false           on
Warning: value must be same on all primary nodes


# bdr.ddl_locking
Ok: all node values are ok


# bdr.ddl_replication
Node               Status  Pending Restart Value Message
----               ------  --------------- ----- -------
bdr-a1             Warning false           0     must be set to on
bdr-a2             Ok      false           on
bdr-b1             Ok      false           on
bdr-b2             Ok      false           on
logical-standby-a1 Ok      false           on
logical-standby-b1 Ok      false           on
subscriber-only-c1 Ok      false           on
witness-a          Ok      false           on
witness-b          Ok      false           on
witness-c          Ok      false           on
Warning: value must be same on all primary nodes
```

```
# bdr.max_writers_per_subscription
Ok: all node values are ok


# bdr.raft_group_max_connections
Ok: all node values are ok


# bdr.replay_progress_frequency
Node               Status  Pending Restart Value Message
----               ------  --------------- ----- -------
bdr-a1             Warning false           61000 must be <= 60000
bdr-a2             Ok      false           60000
bdr-b1             Ok      false           60000
bdr-b2             Ok      false           60000
logical-standby-a1 Ok      false           60000
logical-standby-b1 Ok      false           60000
subscriber-only-c1 Ok      false           60000
witness-a          Ok      false           60000
witness-b          Ok      false           60000
witness-c          Ok      false           60000
Warning: value must be same on all primary nodes


# bdr.role_replication
Node               Status  Pending Restart Value Message
----               ------  --------------- ----- -------
bdr-a1             Warning false           off   must be set to on
bdr-a2             Ok      false           on
bdr-b1             Ok      false           on
bdr-b2             Ok      false           on
logical-standby-a1 Ok      false           on
logical-standby-b1 Ok      false           on
subscriber-only-c1 Ok      false           on
witness-a          Ok      false           on
witness-b          Ok      false           on
witness-c          Ok      false           on
Warning: value must be same on all primary nodes


# bdr.standby_slot_names
Node               Status  Pending Restart Value         Message
----               ------  --------------- -----         -------
bdr-a1             Warning false           bdr_bdrdb_ja… must contain valid logical slots of peer data nodes only
bdr-a2             Warning false           bdr_bdrdb_ja… must contain valid logical slots of peer data nodes only
bdr-b1             Warning false                         must contain valid logical slots of peer data nodes only
bdr-b2             Warning false                         must contain valid logical slots of peer data nodes only
logical-standby-a1 Ok      false
logical-standby-b1 Ok      false
subscriber-only-c1 Ok      false
witness-a          Ok      false
witness-b          Ok      false
witness-c          Ok      false


# bdr.standby_slots_min_confirmed
Node               Status  Pending Restart Value Message
----               ------  --------------- ----- -------
bdr-a1             Warning false           -1    must be >= 1
bdr-a2             Warning false           -1    must be >= 1
bdr-b1             Warning false           -1    must be >= 1
bdr-b2             Warning false           -1    must be >= 1
logical-standby-a1 Ok      false           -1
```

```
logical-standby-b1 Ok      false           -1
subscriber-only-c1 Ok      false           -1
witness-a          Ok      false           -1
witness-b          Ok      false           -1
witness-c          Ok      false           -1


# bdr.start_workers
Ok: all node values are ok


# bdr.xact_replication
Ok: all node values are ok


# max_prepared_transactions
Node               Status  Pending Restart Value Message
----               ------  --------------- ----- -------
bdr-a1             Warning false           16    must be >= 250
bdr-a2             Warning false           16    must be >= 250
bdr-b1             Warning false           16    must be >= 250
bdr-b2             Warning false           16    must be >= 250
logical-standby-a1 Warning false           16    must be >= 250
logical-standby-b1 Warning false           16    must be >= 250
subscriber-only-c1 Warning false           16    must be >= 250
witness-a          Warning false           16    must be >= 250
witness-b          Warning false           16    must be >= 250
witness-c          Warning false           16    must be >= 250


# max_replication_slots
Node               Status   Pending Restart Value Message
----               ------   --------------- ----- -------
bdr-a1             Critical false           8     must be >= 10
bdr-a2             Ok       false           12
bdr-b1             Ok       false           12
bdr-b2             Ok       false           12
logical-standby-a1 Ok       false           12
logical-standby-b1 Ok       false           12
subscriber-only-c1 Ok       false           12
witness-a          Ok       false           12
witness-b          Ok       false           12
witness-c          Ok       false           12
Warning: value must be same on all primary nodes


# max_wal_senders
Ok: all node values are ok


# max_worker_processes
Ok: all node values are ok


# shared_preload_libraries
Node               Status  Pending Restart Value         Message
----               ------  --------------- -----         -------
bdr-a1             Warning false           pg_stat_stat… must contain bdr as first entry
bdr-a2             Warning false           pg_stat_stat… must contain bdr as first entry
bdr-b1             Warning false           pg_stat_stat… must contain bdr as first entry
bdr-b2             Warning false           pg_stat_stat… must contain bdr as first entry
logical-standby-a1 Warning false           pg_stat_stat… must contain bdr as first entry
logical-standby-b1 Warning false           pg_stat_stat… must contain bdr as first entry
subscriber-only-c1 Warning false           pg_stat_stat… must contain bdr as first entry
witness-a          Warning false           pg_stat_stat… must contain bdr as first entry
```

```
witness-b          Warning false           pg_stat_stat… must contain bdr as first entry
witness-c          Warning false           pg_stat_stat… must contain bdr as first entry


# track_commit_timestamp
Ok: all node values are ok


# wal_level
Ok: all node values are ok
```

22 Node types and capabilities

A PGD cluster can contain several different types of node, each with its own role. This section describes the different types of node that can be configured in a PGD cluster.

- Overview is an overview the kinds of node that can exist in PGD clusters and their associated roles.
- Witness nodes looks at the witness node, a special class of PGD node, dedicated to establishing consensus in a group.
- Logical standby nodes shows how to efficiently keep a node on standby synchronized and ready to step in as a primary in the case of failure.
- Subscriber-only nodes and groups looks at how subscriber-only nodes work with subscriber-only groups, how they boost read scalability and the different options for configuring them.

22.1 An overview of PGD Node types

Data nodes

A data node in PGD is a node that runs a Postgres instance. It replicates data to all other data nodes. It also participates in the cluster-wide Raft decision-making around locking and leadership. It can be a member of one or more groups and is, by default, a member of the "top level" group that spans all data nodes in the cluster.

The data node is also the foundation on which the other three nodes are built.

Witness nodes

A witness node behaves like a data node in that it participates in the cluster-wide Raft decision-making around locking and leadership. It doesn't replicate or store data, though. The purpose of a witness node is to be available to ensure that the cluster can achieve a majority it seeks a consensus. Witness nodes has more details.

Logical standby nodes

Logical standby nodes are nodes that receive the logical data changes from another node and replicate them locally. PGD can use a logical standby node to replace the node it's replicating if that node becomes unavailable, with some caveats. See Logical standby nodes for more details.

Subscriber-only nodes

A subscriber-only node is a data node that, as the name suggests, only subscribes to changes in the cluster but doesn't replicate changes to other nodes. You can use subscriber-only nodes as read-only nodes for applications. You create subscriber-only nodes by specifying a data node is `subscriber-only` when you create the node and then adding it to a subscriber-only group. See Subscriber-only nodes and groups for more details.

22.2 Witness nodes

A witness node is a lightweight node that functions as a data node but that doesn't store or replicate data. Use a witness node to allow a PGD cluster that uses Raft consensus to have an odd number of voting nodes and therefore be able to achieve a majority when making decisions.

Witness nodes within PGD groups or regions

One typical use of witness nodes is when a PGD group has two data nodes but resources aren't available for the recommended three data nodes. In this case, you can add a witness node to the PGD group to provide a third voting node to local Raft decision-making. These decisions are primarily about who will be electing a write leader for the proxies to use. With only two nodes, it's possible to have no consensus over which data node is write leader. With two data nodes and a witness, there are two candidates (the data nodes) and three voters (the data nodes and the witness). When a data node is down, then, there are still two voters that can select a write leader.

Witness node outside regions

At a higher level, you can use witness nodes when multiple PGD groups are mapped to different regions. For example, with three data nodes per region in two regions, while running normally, all six data nodes can participate in Raft decisions and obtain DDL and DML global locks. Even when a data node is down, there are sufficient data nodes to obtain a consensus. But if a network partition occurs and connectivity with the other region is lost, then now only three nodes out of six are available, which isn't enough for a consensus. To avoid this scenario, you can deploy a witness node in a third region as part of the PGD cluster. This witness node will allow a consensus to be achieved for most operational requirements of the PGD cluster while a region is unavailable.

22.3 Logical standby nodes

PGD 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.

> **Note**
>
> Logical standby nodes can be used in environments where network traffic between data centers is a concern. Otherwise, having more data nodes per location is always preferred.

Logical standby nodes are nodes that are held in a state of continual recovery, constantly updating until they're required. This behavior is similar to how Postgres physical standbys operate while using logical replication for better performance. `bdr.join_node_group` has the `pause_in_standby` option to make the node stay in halfway-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.

For high availability, 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 PGD 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 PGD 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 PGD peer nodes to replay their progress updates. This behavior 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 PGD cluster by manually performing WAL switches and by requesting all PGD 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 PGD 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.

A logical standby does allow write transactions. 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. PGD 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.

22.4 Subscriber-only nodes and groups

Subscriber-only nodes and groups offer a powerful way to build read scaling into your PGD cluster.

- The Overview introduces how subscriber-only nodes and groups work in PGD.
- Creating a subscriber-only group explains how to create a subscriber-only group and node.
- Joining a node to a subscriber-only group explains how to join a node to an existing subscriber-only group which has members.
- Optimizing subscriber-only groups provides details on how to configure the PGD 5.6 subscriber-only optimized topology feature which uses a group leader for more efficient replication.

22.4.1 An overview of Subscriber-only nodes

Overview

While many use cases rely on accessing a database node which can handle queries and updates, there are also use cases which only require access to a node that can handle read-only database queries. Read scaling like this, by moving the read-only traffic away from active database nodes in the cluster, can improve the performance of the core cluster, whilst making database access more widely available.

Subscriber-only nodes

The basic idea of subscriber-only nodes is to provide a read-only node that you can use to offload read-only queries from the main cluster. The default topology of a PGD cluster is what's called a full mesh topology, where every node connects to every other node. This is the most robust and fault-tolerant way to connect nodes, but it can be inefficient for some use cases.

Each subscriber-only node has to be member of a subscriber-only group, which is a group of nodes that only replicate changes from the rest of the cluster. You can't have a subscriber-only node that's not part of a subscriber-only group.

Subscriber-only groups

Subscriber-only groups in PGD gather together subscriber-only nodes. Each group can address different regions or different application demands.

Unlike data groups, a subscriber-only group has no raft consensus mechanism of its own and no proxies. This also means that a subscriber-only group can have as many subscriber-only nodes as your need.

Previous to PGD 5.6, the existence of a subscriber-only group didn't change the replication topology. All nodes in the subscriber-only group independently received replicated changes from all other nodes in the cluster.

Optimizing subscriber-only groups

In PGD 5.6 and later, you can optionally optimize the topology of subscriber-only groups. For clusters using proxies and raft-enabled groups for their data nodes, subscriber-only groups can use a more efficient model for receiving replicated changes.

The optimized topology option creates a group leader in each subscriber-only group, similar to a write leader in PGD Proxies. The group leader receives all the changes from the cluster and then replicates them to the other nodes in its group. See Optimizing subscriber-only groups for more information on this feature.

22.4.2 Creating Subscriber-only groups and nodes

The process of creating a Subscriber-only node or nodes starts with creating a Subscriber-only group to contain the node or nodes. Perform this step on an existing fully joined node in the PGD cluster.

Creating a Subscriber-only group manually

To create a Subscriber-only group, you must specify the `node_group_type` as `subscriber-only` when creating the group. For example, here we are logged into the node "node-one" running on "host-one". It's a member of it's own data group and as for all nodes, a member of the top-level group, here called `topgroup`. Log into this node directly to create a new Subscriber-only group named `sogroup` with the following SQL command:

```
select bdr.create_node_group('sogroup', 'topgroup', false, 'subscriber-only');
```

or more explicitly with parameter names:

```
select
bdr.create_node_group(node_group_name:='sogroup',
       parent_group_name:='topgroup',
       join_node_group:=false,
       node_group_type:='subscriber-only');
```

This creates a Subscriber-only group named `sogroup` which is a child of the `topgroup` group. The false parameter for `join_node_group` indicates that the node executing this command shouldn't join to the newly created group. Automatically joining the group is the default behavior, which in this case needs to be supressed.

Adding a node to a new Subscriber-only group manually

You can now initialize a new data node and then add it to the Subscriber-only group. Create a data node and configure the bdr extension on it as you would for any other data node. If you deployed manually, see the manual install guide for instructions on how to install and deploy a data node.

You now have to create this new node as a `subscriber-only` node. To do this, log into the new node and run the following SQL command:

```
select bdr.create_node('so-node-1', 'host=so-host-1 dbname=bdrdb port=5444', 'subscriber-
only');
```

Then, log into that new node and add it to the `sogroup` group with the following SQL command:

```
 select bdr.join_node_group('host=host-one dbname=bdrdb port=5444','sogroup');
```

or more explicitly with parameter names:

```
select bdr.join_node_group(dsn:='host=host-one dbname=bdrdb port=5444',
       node_group_name:='sogroup');
```

This instructs the new node to join the `sogroup` group. As it has no knowledge of the cluster topology, it will connect to the node specified in the DSN to receive the necessary information to join the group. In this example, this happens to be the same node as we used to create the subscriber-only group, but it could be any node that's fully joined to the cluster.

22.4.3 Joining nodes to a Subscriber-only group

If you have no subscriber-only groups in your PGD cluster, you must create the groups following the process inCreating Subscriber-only groups and nodes. After you have created a subscriber-only group, you can join subscriber-only nodes to it.

Joining a node to an existing subscriber-only group

Unlike joining a node to a new subscriber-only group, joining a node to an existing subscriber-only group is a simpler process.

First create the new node as a subscriber-only node. Run the following SQL command on the new node:

```
select bdr.create_node('so-node-2', 'host=so-host-2 dbname=bdrdb port=5444', 'subscriber-
only');
```

or more explicitly with parameter names:

```
select bdr.create_node(node_name:='so-node-2',
       dsn:='host=so-host-2 dbname=bdrdb
port=5444',
       node_type:='subscriber-only');
```

This command creates a new node named `so-node-2` on host `so-host-2` and configures it as a subscriber-only node. The node won't be able to join the cluster until joins a group.

In creating a new subscriber-only group, you created a group named `sogroup` and added a subscriber-only node called `so-node-1` on a host `shost-1`. It used a node in an existing data group to facilitate that join. But you can't use this new subscriber-only node to add another subscriber-only node. You must use any active data node that's fully joined to the cluster. In the creating examples, they use `host-one` in the cluster's data group for this task. You can use the following SQL command on `shost-2` to join it to the `sogroup` group:

```
select bdr.join_node_group('host=host-one dbname=bdrdb port=5444','sogroup');
```

or more explicitly with parameter names:

```
select bdr.join_node_group(dsn:='host=host-one dbname=bdrdb port=5444',
       node_group_name:='sogroup');
```

This command instructs the new node to join the `sogroup` group. As it has no knowledge of the cluster topology, it connects to the node specified in the DSN to receive the necessary information to join the group. That node must be fully joined to the cluster as it acts as the source of the request for the new node to join the group.

Once the new node has joined the group, it starts by first synchronizing and then begins to receive replication changes from the other nodes in the cluster.

> **Note**
>
> Unless, the group is using the optimized topology, in which case it replicates changes from a subscriber-only group leader in the subscriber-only group it has joined.

22.4.4 Optimizing subscriber-only groups

With PGD 5.6 and later, it's possible to optimize the topology of subscriber-only groups.

In this optimized topology, a small number of fully active nodes, the write leaders of the data groups, replicate changes to the group leaders of subscriber-only groups, these group leaders then replicate changes to the other members of its subscriber-only group.

Requirements for the optimized topology

You can't enable this model if a cluster has any of the following:

- Data-nodes which are directly members of the top-level group.
- No data-node sub-groups.
- No data-node sub-groups with proxy routing enabled.

If this is the case, the nodes in a subscriber only groups automatically revert to the full mesh topology.

To get the benefit of the new SO group and node replication, you must have your data-nodes in subgroups, with proxy routing enabled on the sub groups.

How the optimized topology works

For clusters using groups for their data nodes, subscriber only groups can use a more efficient model which uses a subscriber-only group leader, similar to a write leader in PGD Proxies.

Each subscriber-only group uses that group leader to replicate changes to other subscriber-only nodes in its group. The group leader acts as a replication proxy for incoming changes.

The write leader nodes in other non-subscriber-only groups replicate changes to the group leader of the subscriber-only group. Other nodes only replicate within their data groups, both within their own group and in other data groups.

Subscriber-only group leaders

With PGD 5.6 and later, each subscriber-only group gets assigned a group leader of its own. This is because subscriber-only groups don't have a group Raft consensus mechanism of their own. Instead, the cluster's top level group uses its Raft consensus mechanism to handle selection of each subscriber-only group's group leader. This group leader selection is on by default on PGD 5.6, regardless of the topology optimization settings.

Group leaders in subscriber-only groups are regularly tested for connectivity and if unavailable, the voting nodes of top-level group select a new subscriber-only node from the subscriber-only group to become group leader. The new group leader is then selected.

With optimized technology turned off, this election has no effect on the replication topology. Without the optimized topology, all data nodes replicate changes to all other nodes in the cluster.

Group leaders in the optimized topology

With the optimized topology enabled, only the subscriber-only group's group leader receives changes from other data group's write leaders in the cluster and it takes on the responsibility of replicating those changes to the other nodes in the subscriber-only group.

The other voting nodes choose the group leader from a subscriber-only group's nodes. Once selected, the whole cluster becomes aware of the change and any data group's write leaders then replicate data only to this newly selected group leader node. Other data nodes in the data groups don't replicate data to the subscriber-only group's nodes.

This approach avoids the explosion of active connections that can happen when there are large numbers of SO nodes and reduces the amount of replication traffic.

The Subscriber-Only node and group form the building block for PGD tree topologies.

Enabling the optimized model

By default, PGD 5.6 forces the full mesh topology. This means the optimization described here is off. To enable the optimized topology, you must have your data-nodes in subgroups, with proxy routing enabled on the sub groups. You can then set the GUC `bdr.force_full_mesh` to `off` to allow the optimization to be activated. Note: This GUC needs to be set in the postgresql.conf file on each data node and each node restarted for the change to take effect.

If any requirements of the optimized topology aren't met, the nodes in a subscriber-only group automatically revert to the full mesh topology. When this happens, you'll find in the logs of the nodes in the cluster messages that explain why the optimization was not possible, such as:

When a data node is part of the top-level node group:

```
node: <nodename> is part of top-level nodegroup: <toplevelgroupname>: changing to full mesh".
```

and when a data group does not have proxy routing enabled:

```
node: <nodename> is in nodegroup: <nodegroupname> that does not have proxy routing: changing to full mesh.
```

23 Commit Scopes

EDB Postgres Distributed (PGD) offers a range of synchronous modes to complement its default asynchronous replication. You use commit scopes to configure these synchronous modes. Commit scopes are rules that define how PGD handles synchronous operations and when the system considers a transaction committed.

Introducing

- Overview introduces the concepts and some of the essential terminology that's used when discussing synchronous commits.
- Durability terminology lists terms used around PGD's durability options, including how to refer to nodes in replication.
- Commit scopes is a more in-depth look at the structure of commit scopes and how to define them for your needs.
- Origin groups introduces the notion of an origin group, and how to leverage these when defining commit scopes rules.
- Commit scope rules looks at the syntax of and how to formulate a commit scope rule.
- Comparing durability options compares how commit scope options behave with regard to durability.
- Degrading commit scope rules shows how to set up a commit scope rule that can gracefully degrade to a lower setting in case of timeouts with a stricter setting.

Commit scope kinds

- Synchronous Commit is a commit scope mechanism that works in a similar fashion to legacy synchronous replication, but from within the commit scope framework.
- Group Commit focuses on the Group Commit option, where you can define a transaction as done when a group of nodes agrees it's done.
- CAMO focuses on the Commit At Most Once option, in which applications take responsibility for verifying that a transaction has been committed before retrying. This ensures that their commits only happen at most once.
- Lag Control looks at the commit scope mechanism which dynamically throttle nodes according to the slowest node and regulates how far out of sync nodes may go when a database node goes out of service.

Working with commit scopes

- Administering addresses how to manage a PGD cluster with Group Commit in use.
- Limitations lists the various combinations of durability options that aren't currently supported or aren't possible. Refer to this before deciding on a durability strategy.
- Legacy synchronous replication shows how you can still access traditional Postgres synchronous operations under PGD.
- Internal timing of operations compares legacy replication with PGD's async and synchronous operations, especially the difference in the order by which transactions are flushed to disk or made visible.

23.1 Overview of durability options

Overview

EDB Postgres Distributed (PGD) allows you to choose from several replication configurations based on your durability, consistency, availability, and performance needs using *commit scopes*.

In its basic configuration, PGD uses asynchronous replication. However, commit scopes can change both the default and the per-transaction behavior.

It's also possible to configure the legacy Postgres synchronous replication using standard `synchronous_standby_names` in the same way as the built-in physical or logical replication. However, commit scopes provide much more flexibility and control over the replication behavior.

The different synchronization settings affect 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.
- Conflict handling: Conflicts can be handled optimistically postcommit, with conflicts resolved when the transaction is replicated based on commit timestamps. Or, they can be handled pessimistically precommit. 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.

Commit scopes allow four kinds of controlling durability of the transaction:

- Synchronous Commit: This kind of commit scope allows for a behavior where the origin node awaits a majority of nodes to confirm and behaves more like a native Postgres synchronous commit.
- Group Commit: This kind of commit scope controls which and how many nodes have to reach a consensus before the transaction is considered to be committable and at what stage of replication it can be considered committed. This option also allows you to control the visibility ordering of the transaction.
- CAMO: This kind of commit scope is a variant of Group Commit, in which the client takes on the responsibility for verifying that a transaction was committed before retrying.
- Lag Control: This kind of commit scope controls how far behind nodes can be in terms of replication before allowing commit to proceed.

Synchronous commit, group commit, and CAMO each support degrading commit scope rules, for even further control of durability.

> **Legacy synchronization availability**
>
> For backward compatibility, PGD still supports configuring synchronous replication with `synchronous_commit` and `synchronous_standby_names`. See Legacy synchronous replication for more on this option. We recommend that you use PGD Synchronous Commit instead.

23.2 Durability terminology

Durability terminology

This page covers terms and definitions directly related to PGD's durability options. For other terms, seeTerminology.

Nodes

PGD nodes take different roles during the replication of a transaction. 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 PGD nodes and responding back to the client with a confirmation or an error.
- The *origin node group* is a PGD group which includes the origin.
- A *partner* node is a PGD node expected to confirm transactions according to Group Commit requirements.
- A *commit group* is the group of all PGD 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.

23.3 Commit scopes

Commit scopes give applications granular control about durability and consistency of EDB Postgres Distributed.

A commit scope is a set of rules that describes the behavior of the system as transactions are committed. The actual behavior depends on which a kind of commit scope a commit scope's rule uses Synchronous Commit, Group Commit, Commit At Most Once, Lag Control, or combination of these.

While most commit scope kinds control the processing of the transaction, Lag Control is the exception as it dynamically regulates the performance of the system in response to replication operations being slow or queued up. It is typically used, though, in combination with other commit scope kinds

Commit scope structure

Every commit scope has a name (a `commit_scope_name`).

Each commit scope has one or more rules.

Each rule within the commit scope has an `origin_node_group` which together uniquely identify the commit scope rule.

The `origin_node_group` is a PGD group and it defines the nodes which will apply this rule when they are the originators of a transaction.

Finally there is the rule which defines what kind of commit scope or combination of commit scope kinds should be applied to those transactions.

So if a commit scope has a rule that reads:

```
origin_node_group := 'example_bdr_group',
rule := 'MAJORITY (example_bdr_group) GROUP COMMIT',
```

Then, the rule is applied when any node in the `example_bdr_group` issues a transaction.

The rule itself specifies how many nodes of a specified group will need to confirm the change - `MAJORITY (example_bdr_group)` - followed by the commit scope kind itself - `GROUP COMMIT`. This translates to requiring that any two nodes in `example_bdr_group` must confirm the change before the change can be considered as comitted.

How a commit scope is selected

When any change takes place, PGD looks up which commit scope is should be used for the transaction or node.

If a transaction specifies a commit scope, that scope will be used.

If not specified, the system will search for a default commit scope. Default commit scopes are a group level setting. The system consults the group tree. Starting at the bottom of the group tree with the node's group and working up, it searches for any group which has a default_commit_scope setting defined. This commit scope will then be used.

If no default_commit_scope is found then the node's GUC, bdr.commit_scope is used. And if that isn't set or is set to `local` then no commit scope applies and PGD's async replication is used.

A commit scope will not be used if it is not local and the node where the commit is being run on is not directly or indirectly related to the origin_node_group.

Creating a Commit Scope

Use `bdr.create_commit_scope` to add our example rule to a commit scope. For example:

```
SELECT
bdr.create_commit_scope(
    commit_scope_name := 'example_scope',
    origin_node_group := 'example_bdr_group',
    rule := 'MAJORITY (example_bdr_group) GROUP
COMMIT',
    wait_for_ready :=
true
);
```

This will add the rule `MAJORITY (example_bdr_group) GROUP COMMIT` for any transaction originating from the `example_bdr_group` to a scope called `example_scope`.

If no rules previously existed in `example_scope`, then adding this rule would make the scope exist.

When a rule is added, the origin_node_group must already exist. If it does not, the whole add operation will be discarded with an error.

The rule will then be evaluated. If the rule mentions groups that don't exist or the settings on the group are incompatible with other configuration setting on the group's nodes, a warning will be emitted, but the rule will be added.

Once the rule is added, the commit scope will be available for use.

The `wait_for_ready` controls whether the `bdr.create_commit_scope()` call blocks until the rule has been added to the relevant nodes. The setting defaults to true and can be omitted.

Using a commit scope

To use our example scope, we can set `bdr.commit_scope` within a transaction

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

You must set the commit scope before the transaction writes any data.

You can set a commit scope as a default for a group or subgroup using `bdr.alter_node_group_option`:

```
SELECT bdr.alter_node_group_option(
  node_group_name := 'example_bdr_group',
  config_key := 'default_commit_scope',
  config_value := 'example_scope'
);
```

To completely clear the default for a group or subgroup, set the `default_commit_scope` value to `local`:

```
SELECT bdr.alter_node_group_option(
  node_group_name := 'example_bdr_group',
  config_key := 'default_commit_scope',
  config_value := 'local'
);
```

You can also make this change using PGD CLI:

```
pgd set-group-options example-bdr-group --option default_commit_scope=example_scope
```

And you can clear the default using PGD CLI by setting the value to `local`:

```
pgd set-group-options example-bdr-group --option default_commit_scope=local
```

Finally, you can set the default commit_scope for a node using:

```
SET bdr.commit_scope =
'example_scope';
```

Set `bdr.commit_scope` to `local` to use the PGD default async replication.

23.4 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. The bottom group of the group tree to which that node belongs is the transaction's *origin group*. To make this transparent for the application, PGD allows a commit scope to define different rules depending on the transaction's origin group.

For example, consider an EDB Postgres Distributed cluster with nodes spread across two data centers: a left (`left_dc`) and a right one (`right_dc`). Assume the top-level PGD 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.create_commit_scope(
    commit_scope_name := 'example_scope',
    origin_node_group := 'left_dc',
    rule := 'ALL (left_dc) GROUP COMMIT (commit_decision=raft) AND ANY 1 (right_dc) GROUP
COMMIT',
    wait_for_ready :=
true
);
SELECT
bdr.create_commit_scope(
    commit_scope_name := 'example_scope',
    origin_node_group := 'right_dc',
    rule := 'ANY 1 (left_dc) GROUP COMMIT AND ALL (right_dc) GROUP COMMIT
(commit_decision=raft)',
    wait_for_ready :=
true
);
```

Now, using the `example_scope` on any node that's part of `left_dc` uses the first scope. Using the same scope on a node that's part of `right_dc` uses the second scope. By combining the `left_dc` and `right_dc` origin rules under one commit scope name, an application can simply use `example_scope` on either data center and get the appropriate behavior for that data center.

Each group can also have a default commit scope specified using the `bdr.alter_node_group_option` admin interface.

Making the above scopes the default ones for all transactions originating on nodes in those groups looks like this:

```
SELECT bdr.alter_node_group_option(
  node_group_name := 'left_dc',
  config_key := 'default_commit_scope',
  config_value := 'example_scope'
);
SELECT bdr.alter_node_group_option(
  node_group_name := 'right_dc',
  config_key := 'default_commit_scope',
  config_value := 'example_scope'
);
```

ORIGIN_GROUP

You can also refer to the origin group of a transaction dynamically when creating a commit scope rule by using `ORIGIN_GROUP`.

This can make certain commit scopes rules like those above in `example_scope`, even easier to specify in that you can simply specify one rule instead of two.

For example, again suppose that for transactions originating from nodes in `right_dc` you want all nodes in `right_dc` to confirm and any 1 from `left_dc` to confirm before the transaction is committed. Also, again suppose that for transactions originating in `left_dc` you want all nodes in `left_dc` and any 1 in `right_dc` to confirm before the transaction is commited. Above we used these two rules for this when defining `example_scope`:

```
SELECT
bdr.create_commit_scope(
    commit_scope_name := 'example_scope',
    origin_node_group := 'left_dc',
    rule := 'ALL (left_dc) GROUP COMMIT (commit_decision=raft) AND ANY 1 (right_dc) GROUP
COMMIT',
    wait_for_ready :=
true
);
SELECT
bdr.create_commit_scope(
    commit_scope_name := 'example_scope',
    origin_node_group := 'right_dc',
    rule := 'ANY 1 (left_dc) GROUP COMMIT AND ALL (right_dc) GROUP COMMIT
(commit_decision=raft)',
    wait_for_ready :=
true
);
```

However, with `ORIGIN_GROUP`, just adding and using the following single-rule commit scope, `example_scope_2`, will have the same effect as the two individual rules we used above in `example_scope`:

```
SELECT
bdr.create_commit_scope(
    commit_scope_name := 'example_scope_2',
    origin_node_group := 'top_group',
    rule := 'ALL ORIGIN_GROUP GROUP COMMIT (commit_decision=raft) AND ANY 1 NOT ORIGIN_GROUP GROUP
COMMIT';
    wait_for_ready :=
true
);
```

Under `example_scope_2`, when a transaction originates from `left_dc`, `ORIGIN_GROUP` maps to `left_dc` and `NOT ORIGIN_GROUP` maps to `right_dc`. Likewise, when a transaction originates from `right_dc`, `ORIGIN_GROUP` maps to `right_dc` and `NOT ORIGIN_GROUP` maps to `left_dc`. So by only specifying one rule, you get the effect of two.

Note that if you added more subgroups, for instance a third child of `top_group`, `middle_dc`, then according to `example_scope_2` above, for transactions originating from `left_dc`, all the nodes in `left_dc` must plus any 1 in `right_dc` and any 1 in `middle_dc` must confirm before the transaction is committed. Of course then for transactions originating in `right_dc` all the nodes in `right_dc` plus any 1 node in `left_dc` and any 1 node in `middle_dc` must confirm before the transaction is committed. Lastly, because `middle_dc` is a child of `top_group`, `example_scope_2` also means that for transactions originating in `middle_dc`, all the nodes in `middle_dc` plus any 1 node in `left_dc` and any 1 node in `right_dc` must confirm before the transaction is committed.

23.5 Commit scope rules

Commit scope rules are at the core of the commit scope mechanism. They define what the commit scope enforces.

Commit scope rules are composed of one or more operations that work in combination. Use an AND between rules.

Each operation is made up of two or three parts: the commit scope group, an optional confirmation level, and the kind of commit scope, which can have its own parameters.

```
commit_scope_group [ confirmation_level ] commit_scope_kind
```

A full formal syntax diagram is available in the Commit scopes reference.

A typical commit scope rule, such as `ANY 2 (group) GROUP COMMIT`, can be broken down into its components. `ANY 2 (group)` is the commit scope group specifying, for the rule, which nodes need to respond and confirm they processed the transaction. In this example, any two nodes from the named group must confirm.

No confirmation level is specified, which means that the default is used. You can think of the rule in full, then, as:

```
ANY 2 (group) ON visible GROUP COMMIT
```

The `visible` setting means the nodes can confirm once all the transaction's changes are flushed to disk and visible to other transactions.

The last part of this operation is the commit scope kind, which in this example is `GROUP COMMIT`. `GROUP COMMIT` is a synchronous two-phase commit that's confirmed when any two nodes in the named group confirm they've flushed the transactions changes and made them visible.

The commit scope group

There are three kinds of commit scope groups: `ANY`, `ALL`, and `MAJORITY`. They're all followed by a list of one or more groups in parentheses. This list of groups combines to make a pool of nodes this operation applies to. This list can be preceded by `NOT`, which inverts the pool to be all other groups that aren't in the list. Witness nodes aren't eligible to be included in this pool, as they don't replicate data.

- `ANY n` is followed by an integer value, `n`. It translates to any `n` nodes in the listed groups' nodes.
- `ALL` is followed by the groups and translates to all nodes in the listed groups' nodes.
- `MAJORITY` is followed by the groups and translates to requiring a half, plus one, of the listed groups' nodes to confirm, to give a majority.
- `ANY n NOT` is followed by an integer value, `n`. It translates to any `n` nodes that aren't in the listed groups' nodes.
- `ALL NOT` is followed by the groups and translates to all nodes that aren't in the listed groups' nodes.
- `MAJORITY NOT` is followed by the groups and translates to requiring a half, plus one, of the nodes that aren't in the listed groups' nodes to confirm, to give a majority.

The confirmation level

PGD nodes can send confirmations for a transaction at different times. In increasing levels of protection, from the perspective of the confirming node, these are:

- `received` — A remote PGD node confirms the transaction immediately after receiving it, prior to starting the local application.
- `replicated` — Confirms after applying changes of the transaction but before flushing them to disk.
- `durable` — Confirms the transaction after all of its changes are flushed to disk.
- `visible` (default) — Confirms 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 parentheses with `ON`, as follows:

- `ANY 2 (right_dc) ON replicated`

- `ALL (left_dc) ON visible` (default)
- `ALL (left_dc) ON received AND ANY 1 (right_dc) ON durable`

> **Note**
>
> If you're familiar with PostgreSQL's `synchronous_standby_names` feature, be aware that while the grammar for `synchronous_standby_names` and commit scopes can look similar, there's a subtle difference. The former doesn't account for the origin node, but the latter does. For example, `synchronous_standby_names = 'ANY 1 (..)'` is equivalent to a commit scope of `ANY 2 (...)`. This difference makes reasoning about majority easier and reflects that the origin node also contributes to the durability and visibility of the transaction.

The commit scope kinds

Currently, there are four commit scope kinds. The following is a summary, with links to more details.

`SYNCHRONOUS_COMMIT`

Synchronous Commit is a commit scope option that's designed to behave like the native Postgres `synchronous_commit` option, but is usable from within the commit scope environment. Unlike `GROUP COMMIT`, it's a synchronous non-two-phase commit operation. Like `GROUP COMMIT`, it supports an optional `DEGRADE ON` clause. The commit scope group that comes before this option controls the groups and confirmation requirements the `SYNCHRONOUS_COMMIT` uses.

For more details, see `SYNCHRONOUS_COMMIT`.

`GROUP COMMIT`

Group Commit is a synchronous, two-phase commit that's confirmed according to the requirements of the commit scope group. `GROUP COMMIT` has options that control:

- Whether to track transactions over interruptions (Boolean, defaults to off)
- How to resolve conflicts (`async` or `eager`, defaults to `async`)
- How to obtain a consensus (`group`, `partner` or `raft`, defaults to `group`)

For more details, see `GROUP COMMIT`.

`CAMO`

Commit At Most Once, or CAMO, allows the client/application, origin node, and partner node to ensure that a transaction is committed to the database at most once. Because the client is involved in the process, an application will require modifications to participate in the CAMO process.

For more details, see `CAMO`.

`LAG CONTROL`

With Lag Control, when the system's replication performance exceeds specified limits, a commit delay can be automatically injected into client interaction with the database, providing a back pressure on clients. Lag Control has parameters to set the maximum commit delay that can be exerted. It also has limits in terms of time to process or queue size that trigger increases in that commit delay.

For more details, see `LAG CONTROL`.

Combining rules

Commit scope rules are composed of one or more operations that work in combination. Use an AND to form a single rule. For example:

```
MAJORITY (Region_A) SYNCHRONOUS_COMMIT AND ANY 1 (Region_A) LAG CONTROL (MAX_LAG_SIZE = '50MB')
```

The first operation sets up a synchronous commit against a majority of `Region_A`. The second operation adds lag control that starts pushing the commit delay up when any one of the nodes in `Region_A` has more than 50MB of lag. This combination of operations allows the lag control to operate when any node is lagging.

23.6 Comparing durability options

Comparison

Most options for synchronous replication available to PGD allow for different levels of synchronization, offering different tradeoffs between performance and protection against node or network outages.

The following list of confirmation levels explains what a user should expect to see when that confirmation level is in effect and how that can affect performance, durability, and consistency.

ON RECEIVED

Expect: The peer node has received the changes. Nothing has been updated in the peer nodes tuple store or written to storage.

Confirmation on reception means that 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.

For: The origin node in the transaction only has to wait for the reception of the transaction. Where transactions are large, it may improve the TPS performance of the system.

Against: An increased likelihood of stale reads. Overall, ON RECEIVED is not robust because data can be lost when either a Postgres server or operating system crash occurs.

ON REPLICATED

Expect: The peer node has received the changes and applied them to the tuple store. The changes have been written to storage, but the storage has not been flushed to disk.

Confirmation on replication means the peer has received and applied the changes. Those changes have been written to storage, but will still be in operating system caches and buffers. The system has yet to persist them to disk.

For: This checkpoint is further down the timeline of transaction processing. The origin node only waits for the transaction to be applied, but not persisted.

Against: There's a slightly lower chance of stale reads over ON RECEIVED. Also, with ON REPLICATED data can survive a Postgres crash but will still not survive an operating system crash.

ON DURABLE

Expect: The peer node has received the changes, applied them to the tuple store and persisted the changes to storage. It has yet to make the changes available to other sessions.

Durable confirmation means that the transaction has been written and flushed to the peer node's storage. This protects against loss of data after a crash and recovery of the peer node. But, if a session commits a transaction with an ON DURABLE rule before disconnecting and reconnecting, the transaction's changes are not guaranteed to be visible to the reconnected session.

When used with the Group Commit commit scope kind, this also means the changes are visible.

For: More robust, able to recover without retransmission in the event of a crash.

Against: Doesn't guarantee consistency in cases of failover.

ON VISIBLE

Expect: The peer node has received and applied the changes, persisted and flushed those changes to storage.

Confirmation of visibility means that the transaction was fully applied remotely. If a session commits a transaction with an ON VISIBLE rule before disconnecting and reconnecting, the transaction's changes are guaranteed to be visible to the reconnected session.

For: Robust and consistent.

Against: Lower performance.

23.7 Degrading commit scope rules

`SYNCHRONOUS_COMMIT`, `GROUP COMMIT`, and `CAMO` each have the optional capability of degrading the requirements for transactions when particular performance thresholds are crossed.

When a node is applying a transaction and that transaction times out, it can be useful to trigger a process of degrading the requirements of the transaction to be completed, rather than just rolling back.

`DEGRADE ON` offers a route for gracefully degrading the commit scope rule of a transaction. At its simplest, `DEGRADE ON` takes a timeout and a second set of commit scope operations that the commit scope can gracefully degrade to.

For instance, after 20ms or 30ms timeout, the requirements for satisfying a commit scope could degrade from `ALL (node_group_name) GROUP COMMIT` to `MAJORITY (node_group_name) GROUP COMMIT`, making the transactions apply more steadily.

You can also require that the write leader be the originator of a transaction in order for the degrade clause to be triggered. This can be helpful in "split brain scenarios" where you have, say, 2 data nodes and a witness node. Supposing there is a network split between the two data nodes and you have connections to both of the data nodes, only one of them will be allowed to degrade, because only one of them will be elected leader through the raft election with the witness node.

Behavior

There are two parts to how the generalized `DEGRADE` clause behaves as it is applied to transactions.

Once during the commit, while the commit being processed is waiting for responses that satisfy the commit scope rule, PGD checks for a timeout and, if the timeout has expired, the commit being processed is reconfigured to wait for the commit scope rule in the `DEGRADE` clause. In fact, by this point, the commit scope rule in the `DEGRADE` clause might already be satisfied.

This mechanism alone is insufficient for the intended behavior, as this alone would mean that every transaction—even those that were certain to degrade due to connectivity issues—must wait for the timeout to expire before degraded mode kicks in, which would severely affect performance in such degrading-cluster scenarios.

To avoid this, the PGD manager process also periodically (every 5s) checks the connectivity and apply rate (the one in bdr.node_replication_rates) and if there are commit scopes that would degrade at that point based on the current state of replication, they will be automatically degraded—such that any transaction using that commit scope when processing after that uses the degraded rule instead of waiting for timeout—until the manager process detects that replication is moving swiftly enough again.

SYNCHRONOUS_COMMIT and GROUP COMMIT

Both `SYNCHRONOUS_COMMIT` and `GROUP COMMIT` have `timeout` and `require_write_lead` parameters, with defaults of `0` and `false` respectively. You should probably always set the `timeout`, as the default of `0` causes an instant degrade. You can also require that the write leader be the originator of the transaction in order to switch to degraded mode (again, default is `false`).

Both `SYNCHRONOUS_COMMIT` and `GROUP COMMIT` also have options regarding which rule you can degrade to—which depends on which rule you are degrading from.

First of all, you can degrade to asynchronous operation:

```
ALL (left_dc) SYNCHRONOUS_COMMIT DEGRADE ON (timeout=20s) TO
ASYNC
```

You can also degrade to a less restrictive commit group with the same commit scope kind (again as long as the kind is either `SYNCHRONOUS_COMMIT` or `GROUP COMMIT`). For instance, you can degrade as follows:

```
ALL (left_dc) SYNCHRONOUS_COMMIT DEGRADE ON (timeout=20s) TO MAJORITY (left_dc)
SYNCHRONOUS_COMMIT
```

or as follows:

```
ANY 3 (left_dc) SYNCHRONOUS_COMMIT DEGRADE ON (timeout=20s) TO ANY 2 (left_dc)
SYNCHRONOUS_COMMIT
```

But you cannot degrade from `SYNCHRONOUS_COMMIT` to `GROUP COMMIT` or the other way around.

CAMO

While CAMO supports both the same `timeout` and `require_write_lead` parameters (with the same defaults, `0` and `false` respectively), the options are simpler in that you can only degrade to asynchronous operation.

```
ALL (left_dc) CAMO DEGRADE ON (timeout=20ms, require_write_lead=true) TO
ASYNC
```

Again, you should set the `timeout` parameter, as the default is `0`.

23.8 Synchronous Commit

Commit scope kind: `SYNCHRONOUS_COMMIT` alias: `SYNCHRONOUS COMMIT`

Overview

PGD's `SYNCHRONOUS_COMMIT` is a commit scope kind that works in a way that's more like PostgreSQL's `synchronous_commit` option in its underlying operation. Unlike the PostgreSQL option, though, it's configured as a commit scope and is easier to configure and interact with in PGD.

Unlike other commit scope kinds, such as `GROUP COMMIT` and `CAMO`, the transactions in a `SYNCHRONOUS_COMMIT` operation aren't transformed into a two-phase commit (2PC) transaction. They work more like a Postgres `synchronous_commit`.

Example

In this example, when this commit scope is in use, any node in the `left_dc` group uses `SYNCHRONOUS_COMMIT` to replicate changes to the other nodes in the `left_dc` group. It looks for a majority of nodes in the `left_dc` group to confirm that they committed the transaction.

```
SELECT bdr.create_commit_scope(
    commit_scope_name := 'example_sc_scope',
    origin_node_group := 'left_dc',
    rule := 'MAJORITY (left_dc) SYNCHRONOUS_COMMIT',
    wait_for_ready := true
);
```

Configuration

`SYNCHRONOUS_COMMIT` supports the optional `DEGRADE ON` clause. See the `SYNCHRONOUS_COMMIT` commit scope reference for specific configuration parameters or see this section regarding Degrade on options.

Confirmation

Confirmation level	PGD Synchronous Commit handling
`received`	A remote PGD node confirms the transaction once it's been fully received and is in the in-memory write queue.
`replicated`	Same behavior as `received`.
`durable`	Confirms the transaction after all of its changes are flushed to disk. Analogous to `synchronous_commit = on` in legacy synchronous replication.
`visible` (default)	Confirms the transaction after all of its changes are flushed to disk and it's visible to concurrent transactions. Analogous to `synchronous_commit = remote_apply` in legacy synchronous replication.

Details

Currently `SYNCHRONOUS_COMMIT` doesn't use the confirmation levels of the commit scope rule syntax.

23.9 Group Commit

Commit scope kind: `GROUP COMMIT`

Overview

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 PGD node to successfully confirm a transaction at COMMIT time. Confirmation can be sent at a number of points in the transaction processing but defaults to "visible" when the transaction has been flushed to disk and is visible to all other transactions.

> **Warning**
>
> Group commit is currently offered as an experimental feature intended for preview and evaluation purposes. While it provides valuable capabilities, it has known limitations and challenges that make it unsuitable for production environments. We recommend that customers avoid using this feature in production scenarios until these limitations are addressed in future releases.

Example

```
SELECT
bdr.create_commit_scope(
    commit_scope_name := 'example_scope',
    origin_node_group := 'left_dc',
    rule := 'ALL (left_dc) GROUP COMMIT(commit_decision=raft) AND ANY 1 (right_dc) GROUP
COMMIT',
    wait_for_ready :=
true
);
```

This example creates a commit scope where all the nodes in the `left_dc` group and any one of the nodes in the `right_dc` group must receive and successfully confirm a committed transaction.

Requirements

During normal operation, Group Commit is transparent to the application. Transactions that were in progress during failover need the reconciliation phase triggered or consolidated by either the application or a proxy in between. This activity currently happens only when either the origin node recovers or when it's parted from the cluster. This behavior is the same as with Postgres legacy built-in synchronous replication.

Transactions committed with Group Commit use two-phase commit underneath. Therefore, configure `max_prepared_transactions` high enough to handle all such transactions originating per node.

Limitations

See the Group Commit section of Limitations.

Configuration

`GROUP_COMMIT` supports optional `GROUP COMMIT` parameters, as well as `ABORT ON` and `DEGRADE ON` clauses. For a full description of configuration parameters, see the GROUP_COMMIT commit scope reference or for more regarding `DEGRADE ON` options in general, see the Degrade options section.

Confirmation

Confirmation level	Group Commit handling
`received`	A remote PGD node confirms the transaction immediately after receiving it, prior to starting the local application.
`replicated`	Confirms after applying changes of the transaction but before flushing them to disk.
`durable`	Confirms the transaction after all of its changes are flushed to disk.
`visible` (default)	Confirms the transaction after all of its changes are flushed to disk and it's visible to concurrent transactions.

Behavior

The behavior of Group Commit depends on the configuration applied by the commit scope.

Commit decisions

You can configure Group Commit to decide commits in three different ways: `group`, `partner`, and `raft`.

The `group` decision is the default. It specifies that the commit is confirmed by the origin node upon receiving as many confirmations as required by the commit scope group. The difference is that the commit decision is made based on PREPARE replication while the durability checks COMMIT (PREPARED) replication.

The `partner` decision is what Commit At Most Once (CAMO) uses. This approach works only when there are two data nodes in the node group. These two nodes are partners of each other, and the replica rather than origin decides whether to commit something. This approach requires application changes to use the CAMO transaction protocol to work correctly, as the application is in some way part of the consensus. For more on this approach, see CAMO.

The `raft` decision uses PGDs built-in Raft consensus for commit decisions. Use of the `raft` decision can reduce performance. It's currently required only when using `GROUP COMMIT` with an ALL commit scope group.

Using an ALL commit scope group requires that the commit decision must be set to `raft` to avoid reconciliation issues.

Conflict resolution

Conflict resolution can be `async` or `eager`.

Async means that PGD does optimistic conflict resolution during replication using the row-level resolution as configured for a given node. This happens regardless of whether the origin transaction committed or is still in progress. See Conflicts for details about how the asynchronous conflict resolution works.

Eager means that conflicts are resolved eagerly (as part of agreement on COMMIT), and conflicting transactions get aborted with a serialization error. This approach provides greater isolation than the asynchronous resolution at the price of performance.

Using an ALL commit scope group requires that the commit decision must be set to `raft` to avoid reconciliation issues.

For details about how Eager conflict resolution works, see Eager conflict resolution.

Aborts

To prevent a transaction that can't get consensus on the COMMIT from hanging forever, the `ABORT ON` clause allows specifying timeout. After the timeout, the transaction abort is requested. If the transaction is already decided to be committed at the time the abort request is sent, the transaction does eventually COMMIT even though the client might receive an abort message.

See also Limitations.

Transaction reconciliation

A Group Commit transaction's commit on the origin node is implicitly converted into a two-phase commit.

In the first phase (prepare), the transaction is prepared locally and made ready to commit. The data is made durable but is uncomitted at this stage, so other transactions can't see the changes made by this transaction. This prepared transaction gets copied to all remaining nodes through normal logical replication.

The origin node seeks confirmations from other nodes, as per rules in the Group Commit grammar. If it gets confirmations from the minimum required nodes in the cluster, it decides to commit this transaction moving onto the second phase (commit). In the commit phase, it also sends this decision by way of replication to other nodes. Those nodes will also eventually commit on getting this message.

There's a possibility of failure at various stages. For example, the origin node may crash after preparing the transaction. Or the origin and one or more replicas may crash.

This leaves the prepared transactions in the system. The `pg_prepared_xacts` view in Postgres can show prepared transactions on a system. The prepared transactions might be holding locks and other resources. To release those locks and resources, either abort or commit the transaction. That decision must be made with a consensus of nodes.

When `commit_decision` is `raft`, then, Raft acts as the reconciliator, and these transactions are eventually reconciled automatically.

When the `commit_decision` is `group`, then, transactions don't use Raft. Instead the write lead in the cluster performs the role of reconciliator. This is because it's the node that's most ahead with respect to changes in its subgroup. It detects when a node is down and initiates reconciliation for such a node by looking for prepared transactions it has with the down node as the origin.

For all such transactions, it sees if the nodes as per the rules of the commit scope have the prepared transaction, it takes a decision. This decision is conveyed over Raft and needs the majority of the nodes to be up to do reconciliation.

This process happens in the background. There's no command for you to use to control or issue this.

Eager conflict resolution

Eager conflict resolution (also known as Eager Replication) prevents conflicts by aborting transactions that conflict with each other with serializable errors during the COMMIT decision process.

You configure it using commit scopes as one of the conflict resolution options for Group Commit.

Usage

To enable Eager conflict resolution, the client needs to switch to a commit scope, which uses it at session level or for individual transactions as shown here:

```
BEGIN;

SET LOCAL bdr.commit_scope =
'eager_scope';

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

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

```
COMMIT;
```

In this case, the `eager_scope` commit scope is defined something like this:

```
SELECT
bdr.create_commit_scope(
    commit_scope_name := 'eager_scope',
    origin_node_group := 'top_group',
    rule := 'ALL (top_group) GROUP COMMIT (conflict_resolution = eager, commit_decision = raft) ABORT ON (timeout
= 60s)',
    wait_for_ready :=
true
);
```

> **Upgrading?**
>
> The old `global` commit scope doesn't exist anymore. The above command creates a scope that's the same as the old `global` scope with `bdr.global_commit_timeout` set to `60s`.

The commit scope group for the Eager conflict resolution rule can only be `ALL` or `MAJORITY`. Where `ALL` is used, the `commit_decision` setting must also be set to `raft`.

Error handling

Given that PGD 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 `ABORT ON 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.

Effects of Eager Replication in general

Increased abort rate

With single-node Postgres, or even with PGD in its default asynchronous replication mode, errors at `COMMIT` time are rare. The added synchronization step due to the use of a commit scope using `eager` for conflict resolution also adds a source of errors. 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.

Effects of MAJORITY and ALL node replication in general

Increased commit latency

Adding a synchronization step due to the use of a commit scope means more communication between the nodes, resulting in more latency at commit time. When `ALL` is used in the commit scope, this also means that the availability of the system is reduced, since any node going down causes transactions to fail.

If one or more nodes are lagging behind, the round-trip delay in getting confirmations can be large, causing high latencies. ALL or MAJORITY node replication adds roughly two network round trips (to the furthest peer node in the worst case). Logical standby nodes and nodes still in the process of joining or catching up 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 setting is independent of the `synchronous_commit` setting.

23.10 Commit At Most Once

Commit scope kind: `CAMO`

Overview

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. However, 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 PGD offers a more general solution and doesn't require an `INSERT`. When activated by `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 PGD node. If the prior transaction status is known, then the application can safely decide whether to retry the transaction.

CAMO works by creating a pair of partner nodes that are two PGD nodes from the same PGD 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.

> **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.

> **Note**
>
> The `CAMO` commit scope kind is mostly an alias for `GROUP COMMIT (transaction_tracking = true, commit_decision = partner)` with an additional `DEGRADE ON` clause.

Requirements

To use CAMO, an application must issue an explicit `COMMIT` message as 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

See the `CAMO` commit scope reference for configuration parameters.

Confirmation

Confirmation Level	CAMO handling
`received`	Not applicable, only uses the default, `VISIBLE`.
`replicated`	Not applicable, only uses the default, `VISIBLE`.
`durable`	Not applicable, only uses the default, `VISIBLE`.
`visible` (default)	Confirms the transaction after all of its changes are flushed to disk and it's visible to concurrent transactions.

Limitations

See the CAMO section of Limitations.

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, a confirmed transaction might 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 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.

Asynchronous mode

When the `DEGRADE ON ... TO ASYNC` clause is used in the commit scope, a node detects whether its CAMO partner is ready. If not, it temporarily switches to asynchronous (local) mode. When in this 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 the `timeout` value of `TO ASYNC`. You can check the current readiness status of a CAMO partner 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 asynchronous mode is only ever triggered by an actual CAMO transaction. This is true either because the commit exceeds the `timeout` value of `TO ASYNC` 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 the current node to be the write lead of a group as configured through the `enable_proxy_routing` node group option. See Commit scopes for syntax. This can prevent a split brain situation due to an isolated node from switching to asynchronous mode. If `enable_proxy_routing` isn't set for the CAMO group, the origin node switches to asynchronous mode immediately.

The switch from asynchronous 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 asynchronous mode there's no added 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, PGD 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 asynchronous network and a relatively quick CAMO partner response.

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

Example

This example considers a setup with two PGD nodes that are the CAMO partner of each other:

```
-- create a CAMO commit scope for a group
 over
-- a definite pair of
nodes
SELECT
bdr.create_commit_scope(
    commit_scope_name := 'example_scope',
    origin_node_group := 'camo_dc',
    rule := 'ALL (left_dc) CAMO DEGRADE ON (timeout=500ms) TO
ASYNC'
);
```

For this CAMO commit scope to be legal, the number of nodes in the group must equal exactly 2. Using ALL or ANY 2 on a group consisting of several nodes is an error because the unquantified group expression doesn't resolve to a definite pair of nodes.

With asynchronous mode

If asynchronous 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 asynchronous mode

If asynchronous 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 another persistence layer. If the application fails, it needs only the information in the database to restart.

To illustrate this, this example shows a retry loop in a CAMO-aware client application, written in a C-like pseudo-code. It expects two DSNs, `origin_dsn` and `partner_dsn`, providing connection information. These usually are the same DSNs as used for the initial call to `bdr.create_node` and can be looked up in `bdr.node_summary`, column `interface_connstr`.

```
PGconn *conn = PQconnectdb(origin_dsn);
```

The process starts connecting to the origin node. Now enter the loop:

```
loop {
    PQexec(conn, "BEGIN");
```

Next, start the transaction and begin populating it with changes:

```
    PQexec(conn, "INSERT INTO ...");
    ...
```

Once you're done, you need to make a record of the local node id and the transaction id. Both are available as parameters.

```
    node_id = PQparameterStatus(conn, "bdr.local_node_id");
    xid = PQparameterStatus(conn, "transaction_id");
```

Now it's ready to try to commit.

```
    PQexec(conn, "COMMIT");
    if (PQresultStatus(res) == PGRES_COMMAND_OK)
        return SUCCESS;
```

If the result is `PGRES_COMMAND_OK`, that's good, and you can move on. But if it isn't, you need to use CAMO to track the transaction to completion. The first question to ask is, "Was the connection bad?"

```
    else if (PQstatus(res) == CONNECTION_BAD)
    {
```

If it was a bad connection, then you can check on the CAMO partner node to see if the transaction made it there.

```
        conn = PQconnectdb(partner_dsn);
        if (!connectionEstablished())
            panic();
```

If you can't connect to the partner node, there's not a lot you can do. In this case, panic, or take similar actions.

But if you can connect, you can use `bdr.logical_transaction_status()` to find out how the transaction did. The code recorded the required values, node_id and xid (the transaction id), just before committing the transaction.

```
        sql = "SELECT bdr.logical_transaction_status($node_id, $xid)";
        txn_status = PQexec(conn, sql);
        if (txn_status == "committed")
            return SUCCESS;
        else
            continue;   // to retry the transaction on the partner
    }
```

If the transaction reports it's been committed, then you can call this transaction a success. No more action is required. If, on the other hand, it doesn't report it's been committed, continue in the loop so the transaction can be retried on the partner node.

```
    else
    {
        if (isPermanentError())
            return FAILURE;
        else
        {
            sleep(increasing_retry_delay);

            continue;
        }
    }
}
```

If status of the transaction wasn't success or bad connection, check if the problem was a permanent error. If so, report a failure of the transaction. If not, you can still retry it. Have the code sleep for a period of time that increases with each retry, and then retry the transaction.

Working with the CAMO partner

> **Permissions required**
>
> A number of the following CAMO functions require permission. Any user wanting to use CAMO must have at least the bdr_application role assigned to them.

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.

To check that the CAMO partner is ready, use the function `bdr.is_camo_partner_ready`. Underneath, this triggers the switch to and from local mode.

To find out more about the configured CAMO partner, use `bdr.get_configured_camo_partner()`. This function returns the local node's CAMO partner.

You can wait on the CAMO partner to process the queue with the function `bdr.wait_for_camo_partner_queue()`. This function is a wrapper of `bdr.wait_for_apply_queue`. The difference is that `bdr.wait_for_camo_partner_queue()` defaults to querying the CAMO partner node. It returns an error if the local node isn't part of a CAMO pair.

To check the status of a transaction that was being committed when the node failed, the application must use the function `bdr.logical_transaction_status()`.

You pass this function the the node_id and transaction_id of the transaction you want to check on. With CAMO used in pair mode, you can 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 all cases, you must call the function within 15 minutes after of issuing the commit. 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 mechanism 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.

Connection pools and proxies

Consider the effect of connection pools and proxies when designing a CAMO cluster. A proxy might freely distribute transactions to all nodes in the commit group, that is, to both nodes of a CAMO pair or to all PGD nodes in case of Eager All-Node Replication.

Take care to ensure that the application fetches the proper node id. When using session pooling, the client remains connected to the same node, so the node id remains constant for the lifetime of the client session. However, with finer-grained transaction pooling, the client needs to fetch the node id for every transaction, as in the example that follows.

A client that isn't directly connected to the PGD nodes might not even notice a failover or switchover. But it can always use the `bdr.local_node_id` parameter to determine the node it's currently connected to. In the crucial situation of a disconnect during COMMIT, the proxy must properly forward that disconnect as an error to the client applying the CAMO protocol.

For CAMO in `received` mode, a proxy that potentially switches between the CAMO pairs must use the `bdr.wait_for_camo_partner_queue` function to prevent stale reads.

CAMO limitations

CAMO limitations are covered in Durability limitations.

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 round-trip 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 added 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 PGD cluster against failure scenarios, such as node crashes or network outages.

23.11 Lag Control

Commit scope kind: `LAG CONTROL`

Overview

Lag Control provides a mechanism where, if replication is running outside of limits set, a delay is injected into the origin node's client connections after processing transactions that make replicable updates. This delay is designed to slow the incoming transactions and bring replication back within the defined limits.

Background

The data throughput of database applications on a PGD origin node can exceed the rate at which committed data can replicate to downstream peer nodes.

If this imbalance persists, it can put satisfying organizational objectives, such as RPO, RCO, and GEO, at risk.

- **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 PGD, RPO determines the acceptable amount of committed data that hasn't been applied to one or more peer nodes.
- **Resource constraint objective (RCO)** acknowledges that finite storage is available. In PGD, the demands on these storage resources increase as lag increases.
- **Group elasticity objective (GEO)** ensures that any node isn't originating new data at a rate that can't be saved to its peer nodes.

To allow organizations to achieve their objectives, PGD offers Lag Control. This feature provides a means to precisely regulate the potential imbalance without intruding on applications. It does so by transparently introducing a delay to READ WRITE transactions that modify data. This delay, the PGD commit delay, starts at 0ms.

Using the LAG CONTROL commit scope kind, you can set a maximum time that commits can be delayed between nodes in a group, maximum lag time, or maximum lag size (based on the size of the WAL).

If the nodes can process transactions within the specified maximums on enough nodes, the PGD commit delay will stay at 0ms or be reduced toward 0ms. If the maximums are exceeded on enough nodes, though, the PGD commit delay on the originating node is increased. It will continue increasing until the Lag Control constraints are met on enough nodes again.

The PGD commit delay happens after a transaction has completed and released all its locks and resources. This timing of the delay allows concurrent active transactions to carry on observing and modifying the delayed transactions values and acquiring its resources.

Strictly speaking, the PGD commit delay isn't a per-transaction delay. It's 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 PGD runtime commit delay to settle within microseconds of the lowest duration possible to maintain a lag measure threshold.

> **PGD commit delay != Postgres commit delay**
>
> Don't conflate the PGD commit delay with the Postgres commit delay. They are unrelated and perform different functions. Don't substitute one for the other.

Requirements

To get started using Lag Control:

- Determine the maximum acceptable commit delay time `max_commit_delay` that all database applications can tolerate.
- Decide on the lag measure to use. Choose either lag size `max_lag_size` or lag time `max_lag_time`.
- Decide on the groups or subgroups involved and the minimum number of nodes in each collection required to satisfy confirmation. This information forms the basis for the definition of a commit scope rule.

Configuration

You specify Lag Control in a commit scope, which allows consistent and coordinated parameter settings across the nodes spanned by the commit scope rule. You can include a Lag Control specification in the default commit scope of a top group or as part of an origin group commit scope.

As in example, take a configuration with two datacenters, `left_dc` and `right_dc`, represented as subgroups:

```
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
);
```

The following code adds Lag Control rules for those two data centers, using individual rules for each subgroup:

```
SELECT
bdr.create_commit_scope(
    commit_scope_name := 'example_scope',
    origin_node_group := 'left_dc',
    rule := 'ALL (left_dc) LAG CONTROL (max_commit_delay=500ms, max_lag_time=30s) AND ANY 1 (right_dc) LAG
CONTROL (max_commit_delay=500ms, max_lag_time=30s)',
    wait_for_ready :=
true
);
SELECT
bdr.create_commit_scope(
    commit_scope_name := 'example_scope',
    origin_node_group := 'right_dc',
    rule := 'ANY 1 (left_dc) LAG CONTROL (max_commit_delay=0.250ms, max_lag_size=100MB) AND ALL (right_dc) LAG
CONTROL (max_commit_delay=0.250ms, max_lag_size=100MB)',
    wait_for_ready :=
true
);
```

You can add a Lag Control commit scope rule to existing commit scope rules that also include Group Commit and CAMO rule specifications.

The `max_commit_delay` is an interval, typically specified in milliseconds (1ms). Using fractional values for sub-millisecond precision is supported.

The `max_lag_size` is an integer that specifies the maximum allowed lag in terms of WAL bytes.

The `max_lag_time` is an interval, typically specified in seconds, that specifies the maximum allowed lag in terms of time.

The maximum commit delay (`max_commit_delay`) is a ceiling value representing a hard limit, which means that a commit delay never exceeds the configured value.

The maximum lag size and time (`max_lag_size` and `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.

Confirmation

Confirmation level	Lag Control handling
`received`	Not applicable, only uses the default, `VISIBLE` .
`replicated`	Not applicable, only uses the default, `VISIBLE` .
`durable`	Not applicable, only uses the default, `VISIBLE` .
`visible` (default)	Not applicable, only uses the default, `VISIBLE` .

Transaction application

The PGD commit delay is applied to all READ WRITE transactions that modify data for user applications. This behavior 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 PGD 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 PGD auxillary processes don't delay at transaction commit. Most notably, PGD writers don't execute a commit delay when applying remote transactions on the local node. This is by design, as PGD 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 is a ceiling value representing a hard limit, which means that a commit delay never exceeds the configured value. Conversely, the maximum lag measures both by size and time and 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 PGD 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 PGD 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 PGD 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 PGD 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 PGD commit delays to allow digestion of the lag contributed by the prior partial bulk load.

Meeting organizational objectives

In the example objectives listed 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 PGD runtime commit delay and the PGD runtime lag measures,

As mentioned, when the maximum PGD runtime commit delay is pegged at the PGD-configured commit-delay limit, and the lag measures consistently exceed their PGD-configured maximum levels, this scenario can be a marker for PGD group expansion.

Lag Control and extensions

The PGD 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 PGD extension is listed before extension-based resource managers in postgresql.conf.

23.12 Administering

When running a PGD cluster with Group Commit, you need to be aware of some things when administering the system, such as how to safely shut down and restart nodes.

Planned shutdown and restarts

When using Group Commit with receive confirmations, take care with planned shutdown or restart. By default, the apply queue is processed 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.

23.13 Legacy synchronous replication using PGD

> **Important**
>
> We highly recommend PGD Synchronous Commit instead of legacy synchronous replication.

Postgres provides physical streaming replication (PSR), which is unidirectional but offers a synchronous variant.

For backward compatibility, PGD still supports configuring synchronous replication with `synchronous_commit` and `synchronous_standby_names`. Consider using Group Commit or Synchronous Commit instead.

Unlike PGD replication options, PSR sync persists first, replicating after the WAL flush of commit record.

Usage

To enable synchronous replication using PGD, you need to add the application name of the relevant PGD peer nodes to `synchronous_standby_names`. The use of `FIRST x` or `ANY x` offers some flexibility if this doesn't conflict with the requirements of non-PGD 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 the application name to `synchronous_standby_names` already enables synchronous replication. Setting `synchronous_commit` to `local` or `off` turns off synchronous replication.

Due to PGD applying the transaction before persisting it, the values `on` and `remote_apply` are equivalent for logical replication.

Comparison

The following table summarizes what a client can expect from a peer node replicated to after receiving 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 PGD, it refers to the `synchronous_commit` setting.

Variant	Mode	Received	Visible	Durable
PSR Async	off (default)	no	no	no
PSR Sync	remote_write (2)	yes	no	no (3)
PSR Sync	on (2)	yes	no	yes
PSR Sync	remote_apply (2)	yes	yes	yes
PGD Legacy Sync (1)	remote_write (2)	yes	no	no
PGD Legacy Sync (1)	on (2)	yes	yes	yes
PGD Legacy Sync (1)	remote_apply (2)	yes	yes	yes

(1) Consider using Group Commit instead.

(2) Unless switched to local mode (if allowed) by setting `synchronous_replication_availability` to `async'`, otherwise the values for the asynchronous PGD default apply.

(3) Written to the OS, durable if the OS remains running and only Postgres crashes.

Postgres configuration parameters

The following table provides an overview of the configuration settings that you must set to a non-default value (req) and those that are optional (opt) but affect a specific variant.

Setting (GUC)	Group Commit	Lag Control	PSR	Legacy Sync
synchronous_standby_names	n/a	n/a	req	req
synchronous_commit	n/a	n/a	opt	opt
synchronous_replication_availability	n/a	n/a	opt	opt

Migration to commit scopes

You configure the Group Commit feature of PGD 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 configuring `synchronous_standby_names` 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.

23.14 Limitations

The following limitations apply to the use of commit scopes and the various durability options they enable.

General limitations

- Legacy synchronous replication uses a mechanism for transaction confirmation different from the one used by CAMO, Eager, and Group Commit. The two aren't compatible, so don't use them together. Whenever you use Group Commit, CAMO, or Eager, make sure none of the PGD nodes are configured in `synchronous_standby_names`.
- Postgres two-phase commit (2PC) transactions (that is, `PREPARE TRANSACTION`) can't be used with CAMO, Group Commit, or Eager because those features use two-phase commit underneath.

Group Commit

Group Commit enables configurable synchronous commits over nodes in a group. If you use this feature, take the following limitations into account:

- Not all DDL can run when you use Group Commit. If you use unsupported DDL, a warning is logged, and the transactions commit scope is set to local. The only supported DDL operations are:
  - Nonconcurrent `CREATE INDEX`
  - Nonconcurrent `DROP INDEX`
  - Nonconcurrent `REINDEX` of an individual table or index
  - `CLUSTER` (of a single relation or index only)
  - `ANALYZE`
  - `TRUNCATE`
- Explicit two-phase commit isn't supported by Group Commit as it already uses two-phase commit.
- Combining different commit decision options in the same transaction or combining different conflict resolution options in the same transaction isn't supported.
- Currently, Raft commit decisions are extremely slow, producing very low TPS. We recommended using them only with the `eager` conflict resolution setting to get the Eager All-Node Replication behavior of PGD 4 and older.

Eager

Eager is available through Group Commit. It avoids conflicts by eagerly aborting transactions that might clash. It's subject to the same limitations as Group Commit.

Eager doesn't allow the `NOTIFY` SQL command or the `pg_notify()` function. It also doesn't allow `LISTEN` or `UNLISTEN`.

CAMO

Commit At Most Once (CAMO) is a feature that aims to prevent applications committing more than once. If you use this feature, take these limitations into account when planning:

- CAMO is designed to query the results of a recently failed COMMIT on the origin node. In case of disconnection, the application must request the transaction status from the CAMO partner. Ensure that you 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 resolve pending notifications on commit. CAMO doesn't allow the `NOTIFY` SQL command or the `pg_notify()` function. They also don't allow `LISTEN` or `UNLISTEN`.

- When replaying changes, CAMO transactions might 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 isn't currently compatible with transaction streaming. Be sure to disable transaction streaming when planning to use CAMO. You can configure this option globally or in the PGD node group. See Transaction streaming configuration.

- CAMO isn't currently compatible with decoding worker. Be sure to not enable decoding worker when planning to use CAMO. You can configure this option in the PGD node group. See Decoding worker disabling.

- Not all DDL can run when you use CAMO. If you use unsupported DDL, a warning is logged and the transactions commit scope is set to local only. The only supported DDL operations are:

    - Nonconcurrent `CREATE INDEX`
    - Nonconcurrent `DROP INDEX`
    - Nonconcurrent `REINDEX` of an individual table or index
    - `CLUSTER` (of a single relation or index only)
    - `ANALYZE`
    - `TRUNCATE`

- Explicit two-phase commit isn't supported by CAMO as it already uses two-phase commit.

- You can combine only CAMO transactions with the `DEGRADE TO` clause for switching to asynchronous operation in case of lowered availability.

23.15 Internal timing of operations

For a better understanding of how the different modes work, it's helpful to know that legacy physical streaming replication (PSR) and PGD apply transactions in different ways.

With Legacy PSR, the order of operations is:

1. Origin flushes a commit record to WAL, making the transaction visible locally.
2. Peer node receives changes and issues a write.
3. Peer flushes the received changes to disk.
4. Peer applies changes, making the transaction visible on the peer.

Note that the change is written to the disk before applying the chnages.

With PGD, by default and with Lag Control, the order of operations is different. In these cases, the change becomes visible on the peer before the transaction is flushed to the peer's disk:

1. Origin flushes a commit record to WAL, making the transaction visible locally.
2. Peer node receives changes into its apply queue in memory.
3. Peer applies changes, making the transaction visible on the peer.
4. Peer persists the transaction by flushing to disk.

For PGD's Group Commit and CAMO, the origin node waits for a certain number of confirmations prior to making the transaction visible locally. The order of operations is:

1. Origin flushes a prepare or precommit record to WAL.
2. Peer node receives changes into its apply queue in memory.
3. Peer applies changes, making the transaction visible on the peer.
4. Peer persists the transaction by flushing to disk.
5. 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
PGD Async	apply first	after WAL flush of commit record
PGD Lag Control	apply first	after WAL flush of commit record
PGD Group Commit	apply first	before COMMIT on origin
PGD CAMO	apply first	before COMMIT on origin

24 Conflict Management

EDB Postgres Distributed 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're events that PGD 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 PGD provides the application a range of choices as to how to resolve them.

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.

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

If you want to avoid conflicts, you can use Group Commit with Eager conflict resolution or conflict-free data types (CRDTs), described in CRDT. You can also use PGD Proxy and route all writes to one write-leader, eliminating the chance for inter-nodal conflicts.

24.1 Conflicts

EDB Postgres Distributed 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 PGD can detect and resolve as they occur. This section introduces the PGD functionality that allows you to manage that detection and resolution.

- Overview introduces the idea of conflicts in PGD and explains how they can happen.
- Types of conflicts lists and discusses the various sorts of conflicts you might run across in PGD.
- Conflict detection introduces the mechanisms PGD provides for conflict detection.
- Conflict resolution explains how PGD resolves conflicts and how you can change the default behavior.
- Conflict logging points out where PGD keeps conflict logs and explains how you can perform conflict reporting.
- Data verification with LiveCompare explains how LiveCompare can help keep data consistent by pointing out conflicts as they arise.

24.1.1 Overview

EDB Postgres Distributed 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 PGD can detect and resolve as they occur. Resolving them depends on the nature of the application and the meaning of the data, so it's important for PGD to provide the application with a range of choices for how to resolve conflicts.

By default, conflicts are resolved at the row level. When changes from two nodes conflict, PGD picks either the local or remote tuple and the discards the other. 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. PGD can detect conflicts and handle them differently for each table using conflict triggers, described in Stream triggers.

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

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. PGD 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.

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 PGD 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. Examples of these mechanisms are `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.

PGD 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, which is lazy replication. Less independence between nodes can avoid conflicts altogether, which is why PGD also offers Eager Replication for when this is important.

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 avoidance happens naturally in many applications. For example, many consumer applications allow only the owning user to change data, such as changing the default billing address on an 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. 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 after 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 and 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 PGD's conflict resolution mechanisms to cope with the conflict.

24.1.2 Types of Conflict

PRIMARY KEY or UNIQUE conflicts

The most common conflicts are row conflicts, where two operations affect a row with the same key in ways they can't on a single node. PGD 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).

PGD handles this situation by retaining the most recently inserted tuple of the two according to the originating node's timestamps. (A user-defined conflict handler can override this behavior.)

This conflict generates the `insert_exists` conflict type, which is by default resolved by choosing the newer row, based on commit time, 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 applying 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.

You can also 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

PGD 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 PGD.

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 the table has multiple rows:

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

Some UPDATE operations succeed:

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

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

Other UPDATE operations 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 PGD.

With PGD, the situation becomes more complex if UPDATE operations 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 the apply on the target fails on both nodes with a duplicate key-value violation error. This error causes the replication to halt and requires manual resolution.

You can avoid this duplicate key violation error, and replication doesn't break, if you set the conflict_type `update_pkey_exists` to `skip`, `update`, or `update_if_newer`. This approach 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 PGD. 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, when an incoming `UPDATE` violates more than one `UNIQUE` index (or the `PRIMARY KEY`), PGD raises a `multiple_unique_conflicts` conflict.

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

UPDATE/DELETE conflicts

One node can update a row that another node deletes at ths same time. 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 database can clean up the deleted row by the time the `UPDATE` is received in case the local node is lagging behind in replication. In this case, PGD can't differentiate between `UPDATE` / `DELETE` conflicts and INSERT/UPDATE conflicts. It 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, leading to the `DELETE` being applied and the row removed. If you enable row version conflict detection, a `delete_recently_updated` conflict is generated. The default resolution for a `delete_recently_updated` conflict is to `skip` the deletion. However, you can configure the resolution or a conflict trigger can be configured to handle it.

INSERT/UPDATE conflicts

When using the default asynchronous mode of operation, a node might receive an `UPDATE` of a row before the original `INSERT` was received. This can happen only when three or more nodes are 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 reconstructing 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).

PGD 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 scenario 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. You can safely ignroe one of them.

Conflicts with three or more nodes

If one node inserts a row that's 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`, which can lead to different data on different nodes. These are divergent conflicts.

This conflict type can happen only with three or more masters. At least two masters 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're 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.

PGD 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 instead logs `delete_missing`.

Future releases will automatically resolve `INSERT` / `DELETE` anomalies by way of rechecks using LiveCompare when `delete_missing` conflicts occur. Applications can perform these manually 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.

PGD has problems with the latter case because PGD 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. PGD 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. If they do, take steps to avoid reuse in 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.

PGD applies changes with `session_replication_role = 'replica'`, so foreign keys aren't rechecked when applying changes. In an active/active environment, this situation can result in FK violations if deletes occur to the referenced table at the same time as inserts into the referencing table. This scenario 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 PGD. 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. The result, then, is a value in the referencing table that's no longer present in the referenced table.

In practice, this situation occurs if the `DELETE` operations occurs on referenced tables in separate transactions from `DELETE` operations on referencing tables, which 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 dangling FKs are possible, the risk of this in general is very low. Thus PGD 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 approach avoids any FK violations at the application level.

The second solution is to add triggers to protect against this case using the PGD-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 approach 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 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.

Data conflicts for roles and tablespace differences

Conflicts can also arise where nodes have global (Postgres-system-wide) data, like roles, that differ. This conflict 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 generates an error in the Postgres logs.

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

Lock conflicts and deadlock aborts

Because PGD 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 PGD 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 PGD 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 PGD 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. However, 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 shouldn't 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 PGD 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 PGD 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 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, PGD reports an error if an `update_missing` conflict is resolved using `insert_or_error` and there are missing TOAST columns.

However, more subtle issues than this one occur in case of concurrent workloads with asynchronous replication. (Eager transactions aren't affected.) Imagine, for example, the following workload on an 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 PGD. (It is when using either built-in logical replication or plain pglogical for multi-master.) PGD 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 PGD 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 PGD AlwaysOn architecture.

> **Note**
>
> Running `VACUUM FULL` or `CLUSTER` on just the TOAST table without doing same on the main table removes metadata needed for the extra logging to work. This means that, for a short period after such a statement, the protection against these concurrency issues isn't 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 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. PGD 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.

24.1.3 Conflict detection

PGD 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 first detected based on whether the replication origin changed, so conflict triggers are called in situations that might not turn out 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.

A node that was offline that reconnects and begins sending data changes 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, PGD 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

PGD 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 thenorigin conflict detection is used.

- Enable `check_full_tuple` or the PGD node group.
- Enable `REPLICA IDENTITY FULL` on all tables that use row version conflict detection.
- Enable row version tracking 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 detection strategy used for a specific table, refer to the column `conflict_detection` of the view `bdr.tables`.

To change the current conflict detection strategy, use bdr.alter_table_conflict_detection.

24.1.4 Conflict resolution

Most conflicts can be resolved automatically. PGD 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`.

PGD lets you override the default behavior of conflict resolution by using bdr.alter_node_set_conflict_resolver.

24.1.5 Conflict logging

To ease diagnosing and handling multi-master conflicts, PGD, by default, logs every conflict into the `bdr.conflict_history` table. You can change this behavior with more granularity using bdr.alter_node_set_log_config.

Conflict reporting

You can summarize conflicts logged to tables 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)
```

24.1.6 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 PGD stack and can be aimed at any pair of PGD nodes. By default, it compares all replicated tables and reports differences. LiveCompare also works with non-PGD 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 in 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 time, allowing for the delays of eventual consistency.

See the LiveCompare documentation for further details.

24.2 Column-level conflict detection

By default, conflicts are resolved at row level. When changes from two nodes conflict, either the local or remote tuple is selected and the other is discarded. For example, commit timestamps for the two conflicting changes might be compared 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.

However, it might sometimes be appropriate to resolve conflicts at the column level rather than the row level, at least in some cases.

- Overview introduces column-level conflict resolution in contrast to row-level conflict resolution, suggesting where it might be a better fit than row-level conflict resolution.
- Enabling and disabling provides an example of enabling column-level conflict resolution and explains how to list tables with column-level conflict resolution enabled.
- Timestamps explicates the difference between using `column_modify_timestamp` and `column_commit_timestamp` and shows how the timestamps associated with column-level conflict resolution can be selected and inspected.

24.2.1 Overview

By default, conflicts are resolved at row level. When changes from two nodes conflict, either the local or remote tuple is selected and the other is discarded. For example, commit timestamps for the two conflicting changes might be compared 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.

However, it might sometimes be appropriate to resolve conflicts at the column level rather than the row level, at least in some cases.

When to resolve at the column level

Consider a simple example in which table t has two integer columns, a and b, and a single row `(1,1)`. On one node execute:

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

On another node, before receiving the preceding `UPDATE`, concurrently execute:

```
UPDATE t SET b =
100
```

> **Note**
>
> 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`.

This sequence results in an `UPDATE-UPDATE` conflict. With the `update_if_newer` conflict resolution, the commit timestamps are compared, and the new row version is kept. Assuming the second node committed last, the result is `(1,100)`, which effectively discards the change to column a.

For many use cases, this behavior is desired and expected. However, for some use cases, 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 conflict and overwrite changes.

For such use cases, it might be more appropriate to resolve conflicts on a given table at the column level. To achieve that, PGD tracks the timestamp of the last change for each column separately and uses that to pick the most recent value, essentially performing `update_if_newer`.

Applied to the previous example, the result is `(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 checks that and fails with an error if this setting is missing.

Special problems for column-level conflict resolution

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.

Handling column-level 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.

24.2.2 Enabling and disabling column-level conflict resolution

> **Permissions required**
>
> Column-level conflict detection uses the `column_timestamps` type. This type requires any user needing to detect column-level conflicts to have at least the bdr_application role assigned.

The bdr.alter_table_conflict_detection() function manages column-level conflict resolution.

Using bdr.alter_table_conflict_detection to enable column-level conflict resolution

The bdr.alter_table_conflict_detection function takes a table name and column name as its arguments. The column is added to the table as a `column_modify_timestamp` column. The function also adds two triggers (BEFORE INSERT and BEFORE UPDATE) that are responsible for maintaining timestamps in the new column before each change.

```
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,
db(# 'column_modify_timestamp', 'cts');
 alter_table_conflict_detection
--------------------------------

t

db=# \d my_app.test_table
                                      Table "my_app.test_table"
 Column |         Type          | Collation | Nullable |
Default
--------+-----------------------+-----------+----------+-------------------------------------------------
 id     | integer               |           | not null |
nextval('my_app.test_table_id_seq'::regclass)
 val    | integer               |           |
|
 cts    | bdr.column_timestamps |           | not null | 's 1 775297963454602 0
0'::bdr.column_timestamps
Indexes:
    "test_table_pkey" PRIMARY KEY, btree
(id)
Triggers:
    bdr_clcd_before_insert BEFORE INSERT ON my_app.test_table FOR EACH ROW EXECUTE FUNCTION
bdr.column_timestamps_current_insert()
    bdr_clcd_before_update BEFORE UPDATE ON my_app.test_table FOR EACH ROW EXECUTE FUNCTION
bdr.column_timestamps_current_update()
```

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

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

Listing tables with column-level conflict resolution

You can list tables having column-level conflict resolution enabled with the following query.

```
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');
```

This query detects the presence of a column of type `bdr.column_timestamp`.

24.2.3 Timestamps in column-level conflict resolution

Column-level conflict resolution depends on a timestamp column being included in the table.

Comparing `column_modify_timestamp` and `column_commit_timestamp`

When you select one of the two column-level conflict detection methods, a column is added to the table that contains a mapping of modified columns and timestamps.

The column that stores timestamp mapping is managed automatically. Don't specify or override the value in your queries, as the results can be unpredictable. When possible, user attempts to override the value are ignored.

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 only conflicts with timestamps in both tuples or in neither of them are seen. 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.

`column_modify_timestamp`

When `column_modify_timestamp` is selected as the conflict detection method, the timestamp assigned to the modified columns is the current timestamp, similar to the value you might get running `select_clock_timestamp()`.

This approach is simple and, for many cases, it's correct, for example, when the conflicting rows modify non-overlapping subsets of columns. Its simplicity can, though, lead to unexpected effects.

For example, if an `UPDATE` affects multiple rows, the clock continues ticking while the `UPDATE` runs. So each row gets a slightly different timestamp, even if they're being modified concurrently by the one `UPDATE`. This behavior, in turn, means that the effects of concurrent changes might get "mixed" in various ways, depending on how the changes performed on different nodes interleaves.

Another possible issue is clock skew. When the clocks on different nodes drift, the timestamps generated by those nodes also drift. This clock skew can induce unexpected behavior such as newer changes being discarded because the timestamps are apparently switched around. However, you can manage clock skew between nodes using the parameters bdr.maximum_clock_skew and bdr.maximum_clock_skew_action.

As the current timestamp is unrelated to the commit timestamp, using it to resolve conflicts means that the result isn't equivalent to the commit order, which means it probably can't be serialized.

When using current timestamps to order changes or commits, the conflicting changes might 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. The node id is used as the tiebreaker in this situation. The higher node id wins. This approach ensures that the same changes are applied on all nodes.

`column_commit_timestamp`

You can also use the actual commit timestamp specified with `column_commit_timestamp` as the conflict detection method. This approach has the advantage of using the commit time, which is the same for all changes made in an `UPDATE`.

> **Note**
>
> Statement transactions might be added in the future, which will address issues with mixing effects of concurrent statements or transactions. Still, neither of these options can ever produce results equivalent to commit order.

Inspecting column timestamps

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

> **Note**
>
> The timestamp mapping is maintained by triggers, and the order in which triggers execute matters. If your custom triggers modify tuples and are executed after the pgl_clcd_ triggers, the modified columns aren't detected correctly. This can lead to incorrect conflict resolution. If you need to modify tuples in your triggers, make sure they're executed before the pgl_clcd_ triggers.

The following functions are useful for inspecting timestamps.

`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)
```

24.3 Conflict-free replicated data types

Conflict-free replicated data types (CRDTs) support merging values from concurrently modified rows instead of discarding one of the rows as the traditional resolution does.

- Overview provides an introduction to CRDTs, including how to use CRDTs in tables, configuration options, and examples of CRDTs.
- Using CRDTs investigates how to use CRDTs in tables, reviews some configuration options, and reviews some examples of CRDTs and how they work.
- Operation-based and state-based CRDTs reviews the differences between operation-based and state-based CRDTs.
- Disk-space requirements covers disk-size considerations for CRDTs, especially state-based CRDTs.
- CRDTs vs conflict handling/reporting explains how conflict handling and reporting works with CRDTs.
- Resetting CRDT values discusses the challenges of resetting CRDT values and provides some guidance on doing so successfully.
- Implemented CRDTs details each of the 6 available CRDTs available in PGD, with implementation examples.

24.3.1 CRDTs Overview

Introduction to CRDTs

Conflict-free replicated data types (CRDTs) support merging values from concurrently modified rows instead of discarding one of the rows as the 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 PGD writer on the apply side without any user action needed.

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

CRDTs in PostgreSQL

The CRDTs 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` . Thus they are available without having to tweak `search_path` .

24.3.2 Using CRDTs

Using CRDTs in tables

> **Permissions required**
>
> PGD CRDTs requires usage access to CRDT types. Therefore, any user needing to access CRDT types must have at least the bdr_application role assigned to them.

To use CRDTs, you need 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:

Non-CRDT example

```
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 different 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.

CRDT example

To use a CRDT counter data type instead, you would follow these steps:

Create the table but with a CRDT (`bdr.crdt_gcounter`) as the counters data type.

```
CREATE TABLE crdt_example
(
    id      integer             PRIMARY KEY,
    counter bdr.crdt_gcounter   NOT NULL DEFAULT 0
);
```

Configure the table for column-level conflict resolution:

```
ALTER TABLE crdt_example REPLICA IDENTITY
FULL;

SELECT bdr.alter_table_conflict_detection('crdt_example',
            'column_modify_timestamp', 'cts');
```

And then insert a row with a value for this example.

```
INSERT INTO crdt_example (id) VALUES (1);
```

If you now issue, as before, the same SQL on two nodes at same time.

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

Once the changes are applied, you find that the counter has managed to concurrenct updates.

```
SELECT id, counter FROM crdt_example WHERE id = 1;
   id |
counter
 -----+-----------
    1 |
2
(1 row)
```

This example shows that the CRDT correctly allows the accumulator columns to work, even in the face of asynchronous concurrent updates that otherwise conflict.

Configuration options

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.

Different types of CRDTs

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.

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

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.

Query planning and optimization

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.

24.3.3 Operation-based and state-based CRDTs

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 with a little bit of code to compute the delta. For example, `crdt_delta_counter` is a domain on a `bigint`.

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 added 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.

24.3.4 CRDT Disk-space requirements

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

Operation-based CRDT disk-space reqs

For operation-based types, this is trivial because the types are merely domains on top of other types. They 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.

State-based CRDT disk-space reqs

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:

- `crdt_gcounter` — `32B (header) + N * 12B (per-node)`
- `crdt_pncounter` -— `48B (header) + N * 20B (per-node)`

  `N` denotes the number of nodes that modified this value.

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.

24.3.5 CRDTs vs conflict handling/reporting

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 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 approach makes the conflict reporting unnecessary.

For this reason, conflict reporting is skipped when the conflict can be fully resolved by CRDT merge. Each column must meet 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**
>
> Conflict reporting is also skipped when there are no CRDT columns but all values in local/remote tuples are equal.

24.3.6 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.

Challenges when resetting CRDT values

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. PGD specifically discourages reusing the same primary key value since it can lead to data anomalies in concurrent cases.

How to reliably handle resetting CRDT values

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.

24.3.7 Implemented CRDTs

Currently, six CRDT data types are 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`).

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` , which is 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` , which is 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;
```

25 Parallel Apply

What is Parallel Apply?

Parallel Apply is a feature of PGD that allows a PGD node to use multiple writers per subscription. This behavior generally increases the throughput of a subscription and improves replication performance.

Configuring Parallel Apply

Two variables control Parallel Apply in PGD 5: `bdr.max_writers_per_subscription` (defaults to 8) and `bdr.writers_per_subscription` (defaults to 2).

```
bdr.max_writers_per_subscription = 8
bdr.writers_per_subscription = 2
```

This configuration gives each subscription two writers. However, in some circumstances, the system might allocate up to eight writers for a subscription.

Changing `bdr.max_writers_per_subscription` requires a server restart to take effect.

You can change `bdr.writers_per_subscription` for a specific subscription without a restart by:

1. Halting the subscription using `bdr.alter_subscription_disable`.
2. Setting the new value.
3. Resuming the subscription using `bdr.alter_subscription_enable`.

First though, establish the name of the subscription using `select * from bdr.subscription`. For this example, the subscription name is `bdr_bdrdb_bdrgroup_node2_node1`.

```
SELECT bdr.alter_subscription_disable
('bdr_bdrdb_bdrgroup_node2_node1');

UPDATE
bdr.subscription
SET num_writers =
4
WHERE sub_name =
'bdr_bdrdb_bdrgroup_node2_node1';

SELECT bdr.alter_subscription_enable
('bdr_bdrdb_bdrgroup_node2_node1');
```

When to use Parallel Apply

Parallel Apply is always on by default and, for most operations, we recommend leaving it on.

Monitoring Parallel Apply

To support Parallel Apply's deadlock mitigation, PGD 5.2 adds columns to `bdr.stat_subscription`. The new columns are `nprovisional_waits`, `ntuple_waits`, and `ncommmit_waits`. These are metrics that indicate how well Parallel Apply is managing what previously would have been deadlocks. They don't reflect overall system performance.

The `nprovisional_waits` value reflects the number of operations on the same tuples being performed by concurrent apply transactions. These are provisional waits that aren't actually waiting yet but could start waiting.

If a tuple's write needs to wait until it can be safely applied, it's counted in `ntuple_waits`. Fully applied transactions that waited before being committed are counted in `ncommit_waits`.

Disabling Parallel Apply

To disable Parallel Apply, set `bdr.writers_per_subscription` to `1`.

Deadlock mitigation

When Parallel Apply is operating, the transactional changes from the subscription are written by multiple writers. However, each writer ensures that the final commit of its transaction doesn't violate the commit order as executed on the origin node. If there's a violation, an error is generated and the transaction can be rolled back.

This mechanism previously meant that when the following are all true, the resulting error could manifest as a deadlock:

- A transaction is pending commit and modifies a row that another transaction needs to change.
- That other transaction executed on the origin node before the pending transaction did.
- The pending transaction took out a lock request.

Additionally, handling the error could increase replication lag due to a combination of the time taken:

- To detect the deadlock
- For the client to roll back its transaction
- For indirect garbage collection of the changes that were already applied
- To redo the work

This is where Parallel Apply's deadlock mitigation, introduced in PGD 5.2, can help. For any transaction, Parallel Apply looks at transactions already scheduled for any row (tuple) that the current transaction wants to write. If it finds one, the row is marked as needing to wait until the other transaction is committed before applying its change to the row. This approach ensures that rows are written in the correct order.

Parallel Apply support

Up to and including PGD 5.1, don't use Parallel Apply with Group Commit, CAMO, and Eager Replication. Disable Parallel Apply in these scenarios. If, using PGD 5.1 or earlier, you're experiencing a large number of deadlocks, you might also want to disable Parallel Apply or consider upgrading.

From PGD 5.2, Parallel Apply works with CAMO. It isn't compatible with Group Commit or Eager Replication, so disable it if Group Commit or Eager Replication are in use.

26 Replication sets

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

Every PGD 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 `bdr.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're 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 PGD administration function calls. Use `bdr.replication_set_add_ddl_filter` to define the commands to replicate.

PGD creates replication set definitions on all nodes. Each node can then be defined to publish or subscribe to each replication set using `bdr.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 PGD nodes in the cluster, as it's also used by default for DDL replication and administration function calls.

Behavior of partitioned tables

PGD 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're replicated like regular tables, that is, to the table of the same name as the partition on the downstream. This behavior 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 PGD doesn't 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 this:

```
 relname | nspname |  conname  |
set_name
---------+---------+-----------+----------
 t2      | public  | t2_x_fkey |
s2
(1 row)
```

This output 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 membership

You can add tables to or remove them from one or more replication sets. Doing so 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 and the node to replicate them to. 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 actions are replicated if the target node is also subscribed to both replication set A and B.

You can control membership using `bdr.replication_set_add_table` and `bdr.replication_set_remove_table`.

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';
```

Behavior with foreign keys shows 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,
       pub_repsets,

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,
          pub_repsets,

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 PGD group replication set. This replication is achieved using a DDL filter with the same name as the PGD group. This filter is added to the default PGD group replication set when the PGD group is created.

You can adjust this behavior 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 `bdr.replication_set_add_ddl_filter` and `bdr.replication_set_remove_ddl_filter` to manipulate DDL filters. They're considered to be `DDL` and are therefore subject to DDL replication and global locking.

Selective replication example

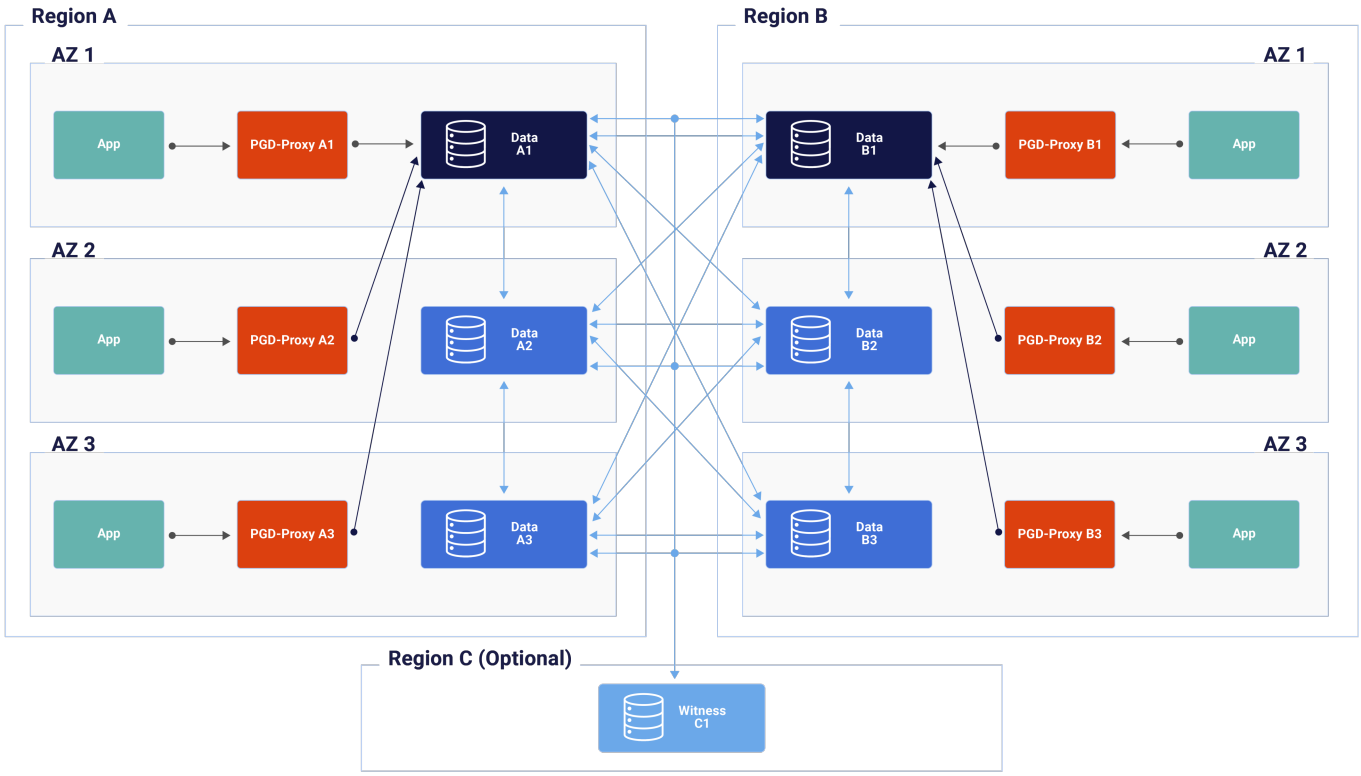
This example configures EDB Postgres Distributed to selectively replicate tables to particular groups of nodes.

Cluster configuration

This example assumes you have a cluster of six data nodes, `data-a1` to `data-a3` and `data-b1` to `data-b3` in two locations. The two locations they're members of are represented as `region_a` and `region_b` groups.

There's also, as we recommend, a witness node named `witness` in `region-c` that isn't mentioned in this example. The cluster is called `sere`.

This configuration looks like this:



This is the standard Always-on Multi-region configuration discussed in Choosing your architecture.

Application requirements

This example works with an application that records the opinions of people who attended performances of musical works. There's a table for attendees, a table for the works, and an opinion table. The opinion table records each work each attendee saw, where and when they saw it, and how they scored the work. Because of data regulation, the example assumes that opinion data must stay only in the region where the opinion was recorded.

Creating tables

The first step is to create appropriate tables:

```
CREATE TABLE attendee
(
   id bigserial PRIMARY KEY,
   email text NOT NULL
);

CREATE TABLE work
(
    id int PRIMARY KEY,
    title text NOT NULL,
    author text NOT
NULL
);

CREATE TABLE opinion
(
    id bigserial PRIMARY KEY,
    work_id int NOT NULL REFERENCES work(id),
    attendee_id bigint NOT NULL REFERENCES
attendee(id),
    country text NOT NULL,
    day date NOT NULL,
    score int NOT NULL
);
```

Viewing groups and replication sets

By default, EDB Postgres Distributed is configured to replicate each table in its entirety to each and every node. This is managed through replication sets.

To view the initial configuration's default replication sets, run:

```
SELECT node_group_name, default_repset,
parent_group_name
FROM bdr.node_group_summary;
```

```
 node_group_name | default_repset | parent_group_name
-----------------+----------------+-------------------
 sere            | sere           |
 region_a        | region_a       | sere
 region_b        | region_b       | sere
 region_c        | region_c       | sere
```

In the output, you can see there's the top-level group, `sere`, with a default replication set named `sere`. Each of the three subgroups has a replication set with the same name as the subgroup. The `region_a` group has a `region_a` default replication set.

By default, all existing tables and new tables become members of the replication set of the top-level group.

Adding tables to replication sets

The next step is to add tables to the replication sets belonging to the groups that represent the regions. As previously mentioned, all new tables are automatically added to the `sere` replication set. You can confirm that by running:

```
SELECT relname, set_name FROM bdr.tables ORDER BY relname,
set_name;
```

```
 relname  | set_name
----------+----------
 attendee | sere
 opinion  | sere
 work     | sere
(3 rows)
```

You want the `opinion` table to be replicated only in `region_a` and, separately, only in `region_b`. To do that, you add the table to the replica sets of each region:

```
SELECT bdr.replication_set_add_table('opinion', 'region_a');
SELECT bdr.replication_set_add_table('opinion', 'region_b');
```

But you're not done, because `opinion` is still a member of the `sere` replication set. When a table is a member of multiple replication sets, it's replicated in each. This doesn't affect performance, though, as each row is replicated only once on each target node. You don't want `opinion` replicated across all nodes, so you need to remove it from the top-level group's replication set:

```
SELECT bdr.replication_set_remove_table('opinion', 'sere');
```

You can now review these changes:

```
SELECT relname, set_name FROM bdr.tables ORDER BY relname,
set_name;
```

```
 relname  | set_name
----------+-------------------
 attendee | sere
 opinion  | region_a
 opinion  | region_b
 work     | sere
(4 rows)
```

This process should provide the selective replication you wanted. To verify whether it did, use the next step to test it.

Testing selective replication

First create some test data: two works and an attendee. Connect directly to `data-a1` to run this next code:

```
INSERT INTO work VALUES (1, 'Aida',
'Verdi');
INSERT INTO work VALUES (2, 'Lohengrin',
'Wagner');
INSERT INTO attendee (email) VALUES
('gv@example.com');
```

Now that there's some data in these tables, you can insert into the `opinion` table without violating foreign key constraints:

```
INSERT INTO opinion (work_id, attendee_id, country, day,
score)
SELECT work.id, attendee.id, 'Italy', '1871-11-19', 3
  FROM work,
attendee
 WHERE work.title = 'Lohengrin'
   AND attendee.email =
'gv@example.com';
```

Once you've done the insert, you can validate the contents of the database on the same node:

```
SELECT a.email
, o.country
, o.day
, w.title
,
w.author
, o.score
FROM opinion
o
JOIN work w ON w.id =
o.work_id
JOIN attendee a ON a.id =
o.attendee_id;
```

```
     email       | country |    day     |   title   | author | score
-----------------+---------+------------+-----------+--------+-------
 gv@example.com | Italy   | 1871-11-19 | Lohengrin | Wagner |     3
(1 row)
```

If you now connect to nodes `data-a2` and `data-a3` and run the same query, you get the same result. The data is being replicated in `region_a`. If you connect to `data-b1`, `data-b2`, or `data-b3`, the query returns no rows. That's because, although the `attendee` and `work` tables are populated, there's no `opinion` row to select. That, in turn, is because the replication of `opinion` on `region_a` happens only in that region.

Now connect to `data-b1` and insert an opinion there:

```
INSERT INTO attendee (email) VALUES
('fb@example.com');

INSERT INTO opinion (work_id, attendee_id, country, day,
score)
SELECT work.id, attendee.id, 'Germany', '1850-08-27', 9
  FROM work,
attendee
 WHERE work.title = 'Lohengrin'
   AND attendee.email =
'fb@example.com';
```

This opinion is replicated only on `region_b`. On `data-b1`, `data-b2`, and `data-b3`, you can run:

```
SELECT a.email
, o.country
, o.day
, w.title
,
w.author
, o.score
FROM opinion
o
JOIN work w ON w.id =
o.work_id
JOIN attendee a ON a.id =
o.attendee_id;
```

```
     email       | country |    day     |   title   | author | score
-----------------+---------+------------+-----------+--------+-------
 fb@example.com | Germany | 1850-08-27 | Lohengrin | Wagner |     9
(1 row)
```

You see the same result on each of the `region_b` data nodes. Run the query on `region_a` nodes, and you don't see this particular entry.

Finally, notice that the `attendee` table is shared identically across all nodes. On any node, run the query:

```
SELECT * FROM attendee;
```

```
         id         |    email
--------------------+----------------
 904252679641903104 | gv@example.com
 904261037006536704 | fb@example.com
(2 rows)
```

27 Stream triggers

PGD 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*.

> **Permissions required**
>
> Stream triggers are a PGD feature that requires permission. Any user wanting to create or drop triggers must have at least the bdr_application role assigned to them.

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 PGD 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's 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's applied while executing a stream trigger isn't replicated to other PGD 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's crash-safe and specific to 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 a 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's 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

PGD uses 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 PGD. They decide what happens when the conflict occurs.

- 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 to delete 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 the changes are applied, 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's 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.

> **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, PGD 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 is 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. This behavior is similar 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're 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's no `TARGET` row.

You can specify multiple transform triggers on each table in PGD. 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 next 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.

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`.

Execution order

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 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. It 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;
$$;
```

28 PGD AutoPartition

PGD AutoPartition allows you to split tables into several partitions. It lets tables grow easily to large sizes using automatic partitioning management. This capability uses features of PGD, 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.

You perform PGD management primarily by using functions that can be called by SQL. All functions in PGD 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

PGD AutoPartition uses the `bdr.autopartition()` function to create or alter the definition of automatic range partitioning for a table. If no definition exists, it's created. Otherwise, later executions will alter the definition.

PGD AutoPartition in PGD 5.5 and later leverages underlying Postgres features that allow a partition to be attached or detached/dropped without locking the rest of the table. Versions of PGD earlier than 5.5 don't support this feature and lock the tables.

An error is raised if the table isn't RANGE partitioned or a multi-column partition key is used.

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

You can change this behavior by setting `managed_locally` to `true`. In that case, all partitions are managed locally on each node. Managing partitions locally is useful when the partitioned table isn't a replicated table. In that case, you might not need or want to have all partitions on all nodes. For example, the built-in `bdr.conflict_history` table isn't a replicated table. It's managed by AutoPartition locally. Each node creates partitions for this table locally and drops them once they're old enough.

Also consider:

- 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`.
- We recommend that you don't manually create or drop partitions for tables managed by AutoPartition. Doing so can make the AutoPartition metadata inconsistent and might cause it to fail.

AutoPartition 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
    );
```

RANGE-partitioned tables

A new partition is added for every `partition_increment` range of values. Lower and upper bound are `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 that's evaluated every time a check is performed. For example, for a partitioned table on column type `date`, suppose `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`. New partitions are then 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 you don't specify the `partition_autocreate_expression` 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. To minimize locking, partitions are dropped at the same time as new partitions are added. If you don't set this value, you must drop the partitions manually.

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

Stopping automatic creation of partitions

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

Waiting for partition creation

Partition creation is an asynchronous process. AutoPartition provides a set of functions to wait for the partition to be created, locally or on all nodes.

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 waits only for the partitions to be created locally. It doesn't guarantee that the partitions also exist on the remote nodes.

To wait for the partition to be created on all PGD 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.

Finding a partition

Use the `bdr.autopartition_find_partition()` function to find the partition for the given partition key value. If a partition to hold that value doesn't exist, then the function returns NULL. Otherwise it returns the Oid of the partition.

Enabling or disabling 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.

29 Explicit two-phase commit (2PC)

> **Note**
>
> Two-phase commit isn't available with Group Commit or CAMO. See Commit scope limitations.

An application can explicitly opt to use two-phase commit with PGD. 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

PGD supports explicit external 2PC using the `PREPARE TRANSACTION` and `COMMIT PREPARED` / `ROLLBACK PREPARED` commands. Externally, an 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 PGD then applies post-commit conflict resolution.

Using `bdr.commit_scope` set to `local` might not seem to make sense with explicit two-phase commit. However, 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';
```

30 Decoding worker

PGD provides an option to enable a decoding worker process that performs decoding once, no matter how many nodes are sent data. This option introduces a new process, the WAL decoder, on each PGD 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 PGD groups and also allow higher replication throughput since the WAL sender process now spends more time on communication.

Enabling

`enable_wal_decoder` is an option for each PGD group, which is currently disabled by default. You can use `bdr.alter_node_group_option()` to enable or disable the decoding worker for a PGD group.

When the decoding worker is enabled, PGD 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 process also needs monitoring. The LCRs that aren't required by any of the PGD 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 0.

Disabling

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 PGD 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 PGD group and set these GUCs to `true` on each of the nodes in the PGD group.

- `bdr.enable_wal_decoder` — When `false`, all WAL senders using LCRs restart to use WAL directly. When `true` along with the PGD group config, a decoding worker process is started to produce LCR and WAL senders that 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 PGD group through a single source. Hence a WAL sender serving a logical standby currently can't use LCRs.
>
> 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.

LCR file names

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 currently all '0'. In the future, they're 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 the 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. You can control the maximum size of an LCR file by adjusting `bdr.max_lcr_segment_file_size`, which defaults to 1 GB.

Using with transaction streaming

It's possible to enable transaction streaming and the decoding worker at the same time. Transaction streaming means that the WAL sender can send a partial transaction before it commits, reducing replication lag. The WAL decoder now supports the decoding of partial transactions, so the decoding worker can decode the partial transaction and store it in an LCR file. The LCR file is then used to apply the transaction on the subscriber node. This in turn reduces CPU usage, by reducing the lag, and reduces disk space usages, since ".spill" files are not generated.

The WAL decoder always streams the transactions to LCRs but based on downstream request the WAL sender either stream transaction or just mimics a normal `BEGIN..COMMIT` scenario.

To support this feature, the system creates additional streaming files. These files have names in that begin with `STR_TXN_<file-name-format>` and `CAS_TXN_<file-name-format>` and each streamed transaction creates their own pair.

To enable transaction streaming with the WAL decoder, set the PGD group's `bdr.streaming_mode` set to 'default' using `bdr.alter_node_group_option`.

31 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.

PGD 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, PGD 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_option()`

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 to 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_option()

You can use the parameter `streaming_mode` in the function `bdr.alter_node_group_option()` 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 are 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 Monitoring PGD writers to check PGD 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)
(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.

32 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 feature to access data on different PGD nodes at a common point in time. For example, you can 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 PGD, this query works only with future timestamps or the special `'current'` value, so you can't use it for historical queries.

PGD works with and improves on that feature in a multi-node environment. First, PGD 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, PGD adds a parameter called `bdr.timestamp_snapshot_keep`. This parameter 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 PGD group. You can use this combination 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.

33 PGD reference

The reference section is a definitive listing of all functions, views, and commands available in EDB Postgres Distributed.

User visible catalogs and views

- bdr.camo_decision_journal
- bdr.commit_scopes
- bdr.conflict_history
- bdr.conflict_history_summary
- bdr.consensus_kv_data
- bdr.crdt_handlers
- bdr.ddl_replication
- bdr.depend
- bdr.global_consensus_journal
- bdr.global_consensus_journal_details
- bdr.global_consensus_response_journal
- bdr.global_lock
- bdr.global_locks
- bdr.group_camo_details
- bdr.group_raft_details
- bdr.group_replslots_details
- bdr.group_subscription_summary
- bdr.group_versions_details
- bdr.leader
- bdr.local_consensus_snapshot
- bdr.local_consensus_state
- bdr.local_node
- bdr.local_node_summary
- bdr.local_sync_status
- bdr.node
- bdr.node_catchup_info
- bdr.node_catchup_info_details
- bdr.node_conflict_resolvers
- bdr.node_group
- bdr.node_group_replication_sets
- bdr.node_group_summary
- bdr.node_local_info
- bdr.node_log_config
- bdr.node_peer_progress
- bdr.node_replication_rates
- bdr.node_slots
- bdr.node_summary
- bdr.queue
- bdr.replication_set
- bdr.replication_set_table
- bdr.replication_set_ddl
- bdr.replication_sets
- bdr.schema_changes
- bdr.sequence_alloc
- bdr.sequences
- bdr.stat_activity
- bdr.stat_commit_scope
- bdr.stat_commit_scope_state
- bdr.stat_raft_followers_state
- bdr.stat_raft_state
- bdr.stat_receiver
- bdr.stat_relation
- bdr.stat_routing_candidate_state
- bdr.stat_routing_state

- bdr.stat_subscription
- bdr.stat_worker
- bdr.stat_writer
- bdr.subscription
- bdr.subscription_summary
- bdr.tables
- bdr.taskmgr_work_queue
- bdr.taskmgr_workitem_status
- bdr.taskmgr_local_work_queue
- bdr.taskmgr_local_workitem_status
- bdr.trigger
- bdr.triggers
- bdr.workers
- bdr.writers
- bdr.worker_tasks

System functions

Version information functions

- bdr.bdr_version
- bdr.bdr_version_num

System information functions

- bdr.get_relation_stats
- bdr.get_subscription_stats

System and progress information parameters

- bdr.local_node_id
- bdr.last_committed_lsn
- transaction_id
- bdr.is_node_connected
- bdr.is_node_ready

Consensus function

- bdr.consensus_disable
- bdr.consensus_enable
- bdr.consensus_proto_version
- bdr.consensus_snapshot_export
- bdr.consensus_snapshot_import
- bdr.consensus_snapshot_verify
- bdr.get_consensus_status
- bdr.get_raft_status
- bdr.raft_leadership_transfer

Utility functions

- `bdr.wait_slot_confirm_lsn`
- `bdr.wait_for_apply_queue`
- `bdr.get_node_sub_receive_lsn`
- `bdr.get_node_sub_apply_lsn`
- `bdr.replicate_ddl_command`
- `bdr.run_on_all_nodes`
- `bdr.run_on_nodes`
- `bdr.run_on_group`
- `bdr.global_lock_table`
- `bdr.wait_for_xid_progress`
- `bdr.local_group_slot_name`
- `bdr.node_group_type`
- `bdr.alter_node_kind`
- `bdr.alter_subscription_skip_changes_upto`

Global advisory locks

- `bdr.global_advisory_lock`
- `bdr.global_advisory_unlock`

Monitoring functions

- `bdr.monitor_group_versions`
- `bdr.monitor_group_raft`
- `bdr.monitor_local_replslots`
- `bdr.wal_sender_stats`
- `bdr.get_decoding_worker_stat`
- `bdr.lag_control`

CAMO functions

- `bdr.is_camo_partner_connected`
- `bdr.is_camo_partner_ready`
- `bdr.get_configured_camo_partner`
- `bdr.wait_for_camo_partner_queue`
- `bdr.camo_transactions_resolved`
- `bdr.logical_transaction_status`

Commit Scope functions

- `bdr.add_commit_scope`
- `bdr.create_commit_scope`
- `bdr.alter_commit_scope`
- `bdr.drop_commit_scope`
- `bdr.remove_commit_scope`

PGD settings

Conflict handling

- bdr.default_conflict_detection

Global sequence parameters

- bdr.default_sequence_kind

DDL handling

- bdr.default_replica_identity
- bdr.ddl_replication
- bdr.role_replication
- bdr.ddl_locking
- bdr.truncate_locking

Global locking

- bdr.global_lock_max_locks
- bdr.global_lock_timeout
- bdr.global_lock_statement_timeout
- bdr.global_lock_idle_timeout
- bdr.lock_table_locking
- bdr.predictive_checks

Node management

- bdr.replay_progress_frequency
- bdr.standby_slot_names

Generic replication

- bdr.writers_per_subscription
- bdr.max_writers_per_subscription
- bdr.xact_replication
- bdr.permit_unsafe_commands
- bdr.batch_inserts
- bdr.maximum_clock_skew
- bdr.maximum_clock_skew_action
- bdr.accept_connections
- bdr.standby_slot_names
- bdr.standby_slots_min_confirmed
- bdr.writer_input_queue_size
- bdr.writer_output_queue_size
- bdr.min_worker_backoff_delay

CRDTs

- `bdr.crdt_raw_value`

Commit scope

- `bdr.commit_scope`

Commit At Most Once

- `bdr.camo_local_mode_delay`
- `bdr.camo_enable_client_warnings`

Transaction streaming

- `bdr.default_streaming_mode`

Lag Control

- `bdr.lag_control_max_commit_delay`
- `bdr.lag_control_max_lag_size`
- `bdr.lag_control_max_lag_time`
- `bdr.lag_control_min_conforming_nodes`
- `bdr.lag_control_commit_delay_adjust`
- `bdr.lag_control_sample_interval`
- `bdr.lag_control_commit_delay_start`

Timestamp-based snapshots

- `bdr.timestamp_snapshot_keep`

Monitoring and logging

- `bdr.debug_level`
- `bdr.trace_level`
- `bdr.track_subscription_apply`
- `bdr.track_relation_apply`
- `bdr.track_apply_lock_timing`

Decoding worker

- `bdr.enable_wal_decoder`
- `bdr.receive_lcr`
- `bdr.lcr_cleanup_interval`

Connectivity settings

- `bdr.global_connection_timeout`
- `bdr.global_keepalives`
- `bdr.global_keepalives_idle`
- `bdr.global_keepalives_interval`
- `bdr.global_keepalives_count`
- `bdr.global_tcp_user_timeout`

Topology settings

- `bdr.force_full_mesh`

Internal settings - Raft timeouts

- `bdr.raft_global_election_timeout`
- `bdr.raft_group_election_timeout`
- `bdr.raft_response_timeout`

Internal settings - Other Raft values

- `bdr.raft_keep_min_entries`
- `bdr.raft_log_min_apply_duration`
- `bdr.raft_log_min_message_duration`
- `bdr.raft_group_max_connections`

Internal settings - Other values

- `bdr.backwards_compatibility`
- `bdr.track_replication_estimates`
- `bdr.lag_tracker_apply_rate_weight`
- `bdr.enable_auto_sync_reconcile`

Node management

- List of node states
- Node-management commands
  - `bdr_init_physical`

Node management interfaces

- `bdr.alter_node_group_option`
- `bdr.alter_node_interface`
- `bdr.alter_node_option`
- `bdr.alter_subscription_enable`
- `bdr.alter_subscription_disable`
- `bdr.create_node`
- `bdr.create_node_group`
- `bdr.drop_node_group`

- `bdr.join_node_group`
- `bdr.part_node`
- `bdr.promote_node`
- `bdr.switch_node_group`
- `bdr.wait_for_join_completion`
- `bdr.alter_node_group_config`

Routing functions

- `bdr.create_proxy`
- `bdr.alter_proxy_option`
- `bdr.drop_proxy`
- `bdr.routing_leadership_transfer`

Commit scopes

- Commit scope syntax
  - commit_scope_degrade_operation
- Commit scope targets
  - ORIGIN_GROUP
- Commit scope groups
  - ANY
  - ANY NOT
  - MAJORITY
  - MAJORITY NOT
  - ALL
  - ALL NOT
- Confirmation level
  - ON received
  - ON replicated
  - ON durable
  - ON visible
- Commit Scope kinds
- SYNCHRONOUS_COMMIT
  - DEGRADE ON parameters
  - commit_scope_degrade_operation
- GROUP COMMIT
  - GROUP COMMIT parameters
  - ABORT ON parameters
  - DEGRADE ON parameters
  - transaction_tracking settings
  - conflict_resolution settings
  - commit_decision settings
  - commit_scope_degrade_operation settings
- CAMO
  - DEGRADE ON parameters
- LAG CONTROL
  - LAG CONTROL parameters

Conflicts

- Conflict detection
  - List of conflict types
- Conflict resolution
  - List of conflict resolvers
  - Default conflict resolvers
  - List of conflict resolutions
- Conflict logging

Conflict functions

- `bdr.alter_table_conflict_detection`
- `bdr.alter_node_set_conflict_resolver`
- `bdr.alter_node_set_log_config`

Replication set management

- `bdr.create_replication_set`
- `bdr.alter_replication_set`
- `bdr.drop_replication_set`
- `bdr.alter_node_replication_sets`

Replication set membership

- `bdr.replication_set_add_table`
- `bdr.replication_set_remove_table`

DDL replication filtering

- `bdr.replication_set_add_ddl_filter`
- `bdr.replication_set_remove_ddl_filter`

Testing and tuning commands

- `pgd_bench`

Global sequence management interfaces

Sequence functions

- `bdr.alter_sequence_set_kind`
- `bdr.extract_timestamp_from_snowflakeid`
- `bdr.extract_nodeid_from_snowflakeid`
- `bdr.extract_localseqid_from_snowflakeid`
- `bdr.timestamp_to_snowflakeid`
- `bdr.extract_timestamp_from_timeshard`
- `bdr.extract_nodeid_from_timeshard`
- `bdr.extract_localseqid_from_timeshard`
- `bdr.timestamp_to_timeshard`

KSUUID v2 functions

- `bdr.gen_ksuuid_v2`
- `bdr.ksuuid_v2_cmp`
- `bdr.extract_timestamp_from_ksuuid_v2`

KSUUID v1 functions

- `bdr.gen_ksuuid`
- `bdr.uuid_v1_cmp`
- `bdr.extract_timestamp_from_ksuuid`

Autopartition

- `bdr.autopartition`
- `bdr.drop_autopartition`
- `bdr.autopartition_wait_for_partitions`
- `bdr.autopartition_wait_for_partitions_on_all_nodes`
- `bdr.autopartition_find_partition`
- `bdr.autopartition_enable`
- `bdr.autopartition_disable`
- Internal functions
- `bdr.autopartition_create_partition`
- `bdr.autopartition_drop_partition`

Stream triggers reference

Stream triggers manipulation interfaces

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

Stream triggers row functions

- `bdr.trigger_get_row`
- `bdr.trigger_get_committs`
- `bdr.trigger_get_xid`
- `bdr.trigger_get_type`
- `bdr.trigger_get_conflict_type`
- `bdr.trigger_get_origin_node_id`
- `bdr.ri_fkey_on_del_trigger`

Stream triggers row variables

- `TG_NAME`
- `TG_WHEN`
- `TG_LEVEL`
- `TG_OP`
- `TG_RELID`
- `TG_TABLE_NAME`
- `TG_TABLE_SCHEMA`
- `TG_NARGS`
- `TG_ARGV[]`

Internal catalogs and views

- `bdr.autopartition_partitions`
- `bdr.autopartition_rules`
- `bdr.ddl_epoch`
- `bdr.event_history`
- `bdr.event_summary`
- `bdr.local_leader_change`
- `bdr.node_config`
- `bdr.node_config_summary`
- `bdr.node_group_config`
- `bdr.node_group_routing_config_summary`
- `bdr.node_group_routing_info`
- `bdr.node_group_routing_summary`
- `bdr.node_routing_config_summary`
- `bdr.proxy_config`
- `bdr.proxy_config_summary`
- `bdr.sequence_kind`

Internal system functions

General internal functions

- `bdr.bdr_get_commit_decisions`
- `bdr.bdr_track_commit_decision`
- `bdr.consensus_kv_fetch`
- `bdr.consensus_kv_store`
- `bdr.decode_message_payload`
- `bdr.decode_message_response_payload`
- `bdr.difference_fix_origin_create`
- `bdr.difference_fix_session_reset`

- bdr.difference_fix_session_setup
- bdr.difference_fix_xact_set_avoid_conflict
- bdr.drop_node
- bdr.get_global_locks
- bdr.get_node_conflict_resolvers
- bdr.get_slot_flush_timestamp
- bdr.internal_alter_sequence_set_kind
- bdr.internal_replication_set_add_table
- bdr.internal_replication_set_remove_table
- bdr.internal_submit_join_request
- bdr.isolation_test_session_is_blocked
- bdr.local_node_info
- bdr.msgb_connect
- bdr.msgb_deliver_message
- bdr.node_catchup_state_name
- bdr.node_kind_name
- bdr.peer_state_name
- bdr.pg_xact_origin
- bdr.request_replay_progress_update
- bdr.reset_relation_stats
- bdr.reset_subscription_stats
- bdr.resynchronize_table_from_node
- bdr.seq_currval
- bdr.seq_lastval
- bdr.seq_nextval
- bdr.show_subscription_status
- bdr.show_workers
- bdr.show_writers

Task manager functions

- bdr.taskmgr_set_leader
- bdr.taskmgr_get_last_completed_workitem
- bdr.taskmgr_work_queue_check_status
- bdr.pglogical_proto_version_ranges
- bdr.get_min_required_replication_slots
- bdr.get_min_required_worker_processes
- bdr.stat_get_activity
- bdr.worker_role_id_name
- bdr.lag_history
- bdr.get_raft_instance_by_nodegroup
- bdr.monitor_camo_on_all_nodes
- bdr.monitor_raft_details_on_all_nodes
- bdr.monitor_replslots_details_on_all_nodes
- bdr.monitor_subscription_details_on_all_nodes
- bdr.monitor_version_details_on_all_nodes
- bdr.node_group_member_info

Conflict functions

- bdr.alter_table_conflict_detection
- bdr.alter_node_set_conflict_resolver
- bdr.alter_node_set_log_config

Column-level conflict functions

- `bdr.column_timestamps_create`

Conflicts

- Conflict detection
  - List of conflict types
- Conflict resolution
  - List of conflict resolvers
  - Default conflict resolvers
  - List of conflict resolutions
- Conflict logging

33.1 User visible catalogs and views

Catalogs and views are listed here in alphabetical order.

`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 (that is, 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.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.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	Subscription that produced this conflict; can be joined to `bdr.subscription` table

Name	Type	Description
origin_node_id	oid	OID (as seen in the pg_replication_origin catalog) of the node that produced the conflicting change
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_change_nr	oid	Index of the change within its transaction
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
origin_node_id	oid	OID (as seen in the pg_replication_origin catalog) of the node that produced the conflicting change
remote_commit_lsn	pg_lsn	Commit LSN of the transaction which produced the conflicting change on the remote node (an origin)
remote_change_nr	oid	Index of the change within its transaction
local_time	timestamp with time zone	Local time of the conflict
local_tuple_commit_time	timestamp with time zone	Time of local commit
remote_commit_time	timestamp with time zone	Time of remote commit
conflict_type	text	Type of conflict
conflict_resolution	text	Resolution adopted

`bdr.consensus_kv_data`

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

`bdr.consensus_kv_data` Columns

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

`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	Command tags it applies on (regular expression)
set_ddl_role	text	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 PGD 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
node_group_name	name	Name of the node group
log_index	int8	ID of the journal entry
term	int8	Raft term
request_id	int8	ID of the request
origin_id	oid	ID of the node where the request originated
req_payload	bytea	Payload of the request
origin_node_name	name	Name of the node where the request originated
message_type_no	oid	ID of the PGD message type for the request
message_type	text	Name of the PGD message type for the request
message_payload	text	PGD message payload for the request
response_message_type_no	oid	ID of the PGD message type for the response
response_message_type	text	Name of the PGD message type for the response
response_payload	text	PGD 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 PGD 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)
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 PGD's shared-memory lock state tracking, giving administrators greater insight into PGD'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	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, that is, 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.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)

Name	Type	Description
receive_lsn	text	Latest LSN of any change or message received (can go backwards in case of restarts)
apply_queue_size	text	Bytes difference between apply_lsn and receive_lsn

`bdr.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
node_group_name	name	Name of the group is part of
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 PGD slot information from all nodes.

`bdr.group_replslots_details` columns

Name	Type	Description
node_group_name	text	Name of the PGD group
origin_name	text	Name of the origin node
target_name	text	Name of the target node
slot_name	text	Slot name on the origin node used by this subscription
active	text	Is the slot active (does it have a connection attached to it)
state	text	State of the replication (catchup, streaming, ...) or 'disconnected' if offline
write_lag	interval	Approximate lag time for reported write
flush_lag	interval	Approximate lag time for reported flush
replay_lag	interval	Approximate lag time for reported replay
sent_lag_bytes	int8	Bytes difference between sent_lsn and current WAL write position
write_lag_bytes	int8	Bytes difference between write_lsn and current WAL write position
flush_lag_bytes	int8	Bytes difference between flush_lsn and current WAL write position
replay_lag_byte	int8	Bytes difference between replay_lsn and current WAL write position

`bdr.group_subscription_summary`

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

`bdr.group_subscription_summary` columns

Name	Type	Description
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 PGD 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	PGD version on the node

`bdr.leader`

Tracks leader nodes across subgroups in the cluster. Shows the status of all write leaders and subscriber-only group leaders (when optimized topology is enabled) in the cluster.

`bdr.leader` columns

Name	Type	Description
node_group_id	oid	ID of the node group.
leader_node_id	oid	ID of the leader node.
generation	int	Generation of the leader node. Leader_kind sets semantics.
leader_kind	"char"	Kind of the leader node.

Leader_kind values can be:

Value	Description
W	Write leader, as per proxy routing. In this case leader is maintained by subgroup Raft instance. `generation` corresponds to `write_leader_version` of respective `bdr.node_group_routing_info` record.
S	Subscriber-only group leader. This designated member of a SO group subscribes to upstream data nodes and is tasked with publishing upstream changes to remaining SO group members. Leader is maintained by top-level Raft instance. `generation` is updated sequentially upon leader change.

`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` (plus `pub_repsets` and `sub_repsets`), 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	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.node`

This table lists all the PGD nodes in the cluster.

The view `bdr.node_summary` provides a human-readable version of most of the columns from `bdr.node`.

`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
generation	smallint	Counter incremented when a node joins with the same name as a previous node
node_kind	oid	ID of the node kind
node_join_finished	boolean	Check if the join is finished

`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_catchup_info_details`

A view of `bdr.node_catchup_info` catalog which shows info in more friendly way

`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 PGD node groups. See also `bdr.node_group_summary` for a view containing user-readable details.

`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	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
node_group_default_commit_scope	oid	ID of the node group's default commit scope
node_group_location	char	Name of the location associated with the node group
node_group_enable_proxy_routing	char	Whether the node group allows routing from `pgd-proxy`
node_group_enable_raft	bool	Whether the node group allows Raft Consensus

`bdr.node_group_replication_sets`

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

`bdr.node_group_replication_sets` columns

Name	Type	Description
node_group_name	name	Name of the PGD 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_group_summary`

A view containing user-readable details about node groups. See also `bdr.node_group`.

`bdr.node_group_summary` columns

Name	Type	Description
node_group_name	name	Name of the node group
default_repset	name	Default replication set for this node group
parent_group_name	name	Name of parent group (NULL if this is a root group)
node_group_type	text	Type of the node group (one of "global", "data", "shard" or "subscriber-only")
apply_delay	interval	How long a subscriber waits before applying changes from the provider
check_constraints	boolean	Whether the apply process checks constraints when applying data
num_writers	integer	Number of writers to use for subscriptions backing this node group
enable_wal_decoder	boolean	Whether the group has enable_wal_decoder set
streaming_mode	text	Transaction streaming setting: "off", "file", "writer", "auto" or "default"
default_commit_scope	name	Name of the node group's default commit scope
location	name	Name of the location associated with the node group
enable_proxy_routing	boolean	Whether the node group allows routing from `pgd-proxy`
enable_raft	boolean	Whether the node group allows Raft Consensus
route_writer_max_lag	bigint	Maximum write lag accepted
route_reader_max_lag	bigint	Maximum read lag accepted
route_writer_wait_flush	boolean	Switch if we need to wait for the flush

`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_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, that is, 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	OID of the originating node that reported this position info
peer_node_id	oid	OID of the node's peer (remote node) for which this position info was reported
last_update_sent_time	timestamptz	Time at which the report was sent by the originating node
last_update_recv_time	timestamptz	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_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	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 PGD.

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 PGD group
node_group_id	oid	OID of the PGD group
origin_name	name	Name of the origin node
target_name	name	Name of the target node
origin_id	oid	OID of the origin node
target_id	oid	OID of the target node
local_slot_name	name	Name of the replication slot according to PGD
slot_name	name	Name of the slot according to Postgres (same as above)
is_group_slot	boolean	True if the slot is the node-group crash recovery slot for this node (see ["Group Replication Slot"](nodes#Group Replication Slot))
is_decoder_slot	boolean	Is this slot used by the decoding worker feature
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	OID of the current database

Name	Type	Description
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	PID of the process attached to the slot
xmin	xid	XID needed by the slot
catalog_xmin	xid	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
usename	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 PGD 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 PGD group the node is part of

Name	Type	Description
interface_connstr	text	Connection string to the node
peer_state_name	text	Consistent state of the node in human readable form
peer_target_state_name	text	State 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
node_id	oid	OID of the node
node_group_id	oid	OID of the PGD node group
node_kind_name	oid	Node kind name

`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	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	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	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 PGD group, even if they aren't currently used by any node.

`bdr.replication_sets` columns

Name	Type	Description
set_id	oid	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 PGD.

`bdr.schema_changes` columns

Name	Type	Description
schema_changes_ts	timestampstz	ID of the trigger
schema_changes_change	char	Flag of change type
schema_changes_classid	oid	Class ID
schema_changes_objectid	oid	Object ID
schema_changes_subid	smallint	Subscription
schema_changes_descr	text	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	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 PGD 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 PGD.

`bdr.stat_commit_scope`

A view containing statistics for each commit scope.

`bdr.stat_commit_scope` columns

Column	Type	Description
commit_scope_name	name	Name of the commit scope
group_name	name	Name of group for which the commit scope is defined
ncalls	bigint	The number of times the commit scope was used
ncommits	bigint	The number of successful commits were made with the commit scope
naborts	bigint	The number of times the commit scope used was eventually aborted
total_commit_time	double precision	Total time spent committing using the commit scope, in milliseconds
min_commit_time	double precision	Minimum time spent committing using the commit scope, in milliseconds
max_commit_time	double precision	Maximum time spend committing using the commit scope, in milliseconds
mean_commit_time	double precision	Mean time spent committing using the commit scope, in milliseconds
stats_reset	timestamp with time zone	Time at which all statistics in the view were last reset

`bdr.stat_commit_scope_state`

A view of information about the current use of commit scopes by backends.

`bdr.stat_commit_scope_state` columns

Column	Type	Description
pid	integer	Process ID of the backend
commit_scope_name	name	Name of the commit scope being used
group_name	name	Name of group for which the commit scope is defined
waiting_op_num	integer	Index of the first operation in the commit scope that is not satisfied yet
waiting_prepare_confirmations	integer	The number of PREPARE confirmations that are still needed by the operation
waiting_commit_confirmations	integer	The number of COMMIT confirmations that are still needed by the operation
waiting_lsn_confirmations	integer	The number of LSN confirmations that are still needed by the operation

`bdr.stat_raft_followers_state`

A view of the state of the raft leader's followers on the Raft leader node (empty on other nodes).

`bdr.stat_raft_followers_state` columns

Column	Type	Description
group_name	name	The group this information is for (each group can have a separate consensus configured)
node_name	name	Name of the follower node
sent_commit_index	bigint	Latest Raft index sent to the follower node
match_index	bigint	Raft index we expect to match the next response from the follower node
last_message_time	timestamp with time zone	Last message (any, including requests) seen from the follower node
last_heartbeat_send_time	timestamp with time zone	Last time the leader sent heartbeat to the follower node
last_heartbeat_response_time	Lasat time the leader has seen a heartbeat response from the follower node	
approx_clock_drift_ms	bigint	Approximate clock drift seen by the leader against the follower node in milliseconds

`bdr.stat_raft_state`

A view describing the state of the Raft consensus on the local node.

`bdr.stat_raft_state` columns

Column	Type	Description
group_name	name	The group this information is for (each group can have a separate consensus configured)
raft_stat	text	State of the local node in the Raft ('LEADER', 'CANDIDATE', 'FOLLOWER', 'STOPPED')
leader_name	name	Name of the Raft leader, if any
voted_for_name	name	The node the local node voted for as leader last vote
is_voting	boolean	The local node part of Raft is voting
heartbeat_timeout_ms	bigint	The heartbeat timeout on the local node
heartbeat_elapsed_ms	bigint	The number of milliseconds that have elapsed since the local node has seen a heartbeat from the leader
current_term	bigint	The current Raft term the local node is at
commit_index	bigint	The current Raft commit index the local node is at
apply_index	bigint	The Raft commit index the local node applied to catalogs
last_log_term	bigint	Last Raft term in the request log
last_log_index	bigint	Last Raft index in the request log
oldest_log_index	bigint	Oldest Raft index still in the request log
newest_prunable_log_index	bigint	Newest Raft index that can be safely removed from the request log
snapshot_term	bigint	Raft term of the last snapshot
snapshot_index	bigint	Raft index of the last snapshot
nnodes	integer	Number of nodes in the Raft consensus (should normally be the same as the number of nodes in the group)
nvoting_nodes	integer	Number of voting nodes in the Raft consensus

`bdr.stat_receiver`

A view containing all the necessary info about the replication subscription receiver processes.

`bdr.stat_receiver` columns

Column	Type	Description
worker_role	text	Role of the BDR worker (always 'receiver')
worker_state	text	State of receiver worker (can be 'running', 'down', or 'disabled')
worker_pid	integer	Process id of the receiver worker
sub_name	name	Name of the subscription the receiver belongs to
sub_slot_name	name	Replication slot name used by the receiver
source_name	name	Source node for this receiver (the one it connects to), this is normally the same as the origin node, but is different for forward mode subscriptions
origin_name	name	The origin node for this receiver (the one it receives forwarded changes from), this is normally the same as the source node, but is different for forward mode subscriptions
subscription_mode	char	Mode of the subscription, see `bdr.subscription_summary` for more details
sub_replication_sets	text[]	Replication sets this receiver is subscribed to
sub_apply_delay	interval	Apply delay interval
receive_lsn	pg_lsn	LSN of the last change received so far
receive_commit_lsn	pg_lsn	LSN of the last commit received so far
xact_apply_lsn	pg_lsn	Last applied transaction LSN
xact_flush_lsn	pg_lsn	Last flushed transaction LSN
xact_apply_timestamp	timestamp with time zone	Last applied transaction (commit) timestamp
worker_start	timestamp with time zone	Time at which the receiver started
worker_xact_start	timestamp with time zome	Time at which the receiver started local db transaction (if it is currently processing a local transaction), usually NULL, see `xact_start` in `pg_stat_activity` for more details
worker_backend_state_change	timestamp with time zone	Backend state change timestamp, see `state_change` in `pg_stat_activity` for more details
worker_backend_state	text	Current backend state, see `state` in `pg_stat_activity` for more details
wait_event_type	text	Type of wait event the receiver is currently waiting on (if any), see `wait_event_type` in `pg_stat_activity` for more details
wait_event	text	Exact event the receiver is currently waiting on (if any, see `wait_event` in `pg_stat_activity` for more details)

`bdr.stat_relation`

Shows apply statistics for each relation. Contains data only if tracking is enabled with `bdr.track_relation_apply` and if data was replicated for a given relation.

`lock_acquire_time` is updated only if `bdr.track_apply_lock_timing` is set to `on` (default: `off`).

You can reset the stored relation statistics by calling `bdr.reset_relation_stats()`.

`bdr.stat_relation` columns

Column	Type	Description
nspname	name	Name of the relation's schema
relname	name	Name of the relation

Column	Type	Description
relid	oid	OID of the relation
total_time	double precision	Total time spent processing replication for the relation, in milliseconds
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)
stats_reset	timestamp with time zone	Time of the last statistics reset (performed by `bdr.reset_relation_stats()`)

`bdr.stat_routing_candidate_state`

A view of information about the routing candidate nodes on the Raft leader (empty on other nodes).

`bdr.stat_routing_candidate_state` columns

Column	Type	Description
node_group_name	name	The group this information is for (each group can have a separate routing proxy)
node_name	name	Candidate node name
node_route_fence	boolean	The node is fenced (when true it cannot become leader or read-only connection target)
node_route_reads	boolean	The node is being considered as a read-only connection target
node_route_writes	boolean	The node is being considered as a write lead candidate.
last_message_time	timestamp with time zone	The time of the last Raft message (any, including requests) seen by this node (used to check liveness of node)

`bdr.stat_routing_state`

A view of the state of the connection routing which PGD Proxy uses to route the connections.

`bdr.stat_routing_state` columns

Column	Type	Description
node_group_name	name	The group this is information for (each group can have a separate routing proxy)
write_lead_name	name	Name of the write lead node
previous_write_lead_name	name	Name of the previous write lead node
read_names	name[]	Array of nodes to which read-only connections are routed

Column	Type	Description
write_candidate_names	name[]	Nodes that match all criteria needed to become write lead in case of failover
read_candidate_names	name[]	Nodes that match all criteria needed to become read-only connection targets in case of failover

`bdr.stat_subscription`

Shows apply statistics for each subscription. Contains data only if tracking is enabled with `bdr.track_subscription_apply`.

You can reset the stored subscription statistics by calling `bdr.reset_subscription_stats()`.

`bdr.stat_subscription` columns

Column	Type	Description
sub_name	name	Name of the subscription
subid	oid	OID of the subscription
mean_apply_time	double precision	Average time per apply transaction, in milliseconds
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)
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 STREAM START messages processed
nstream_stop	bigint	Number of STREAM STOP messages processed
nstream_commit	bigint	Number of streaming transactions committed
nstream_abort	bigint	Number of streaming transactions aborted
nstream_prepare	bigint	Number of streaming transactions prepared
nstream_insert	bigint	Number of streaming inserts processed
nstream_update	bigint	Number of streaming updates processed
nstream_delete	bigint	Number of streaming deletes processed
nstream_truncate	bigint	Number of streaming truncates 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

Column	Type	Description
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
npre_commit_confirmations	bigint	Number of precommit confirmations by CAMO partners
npre_commit	bigint	Number of precommits
ncommit_prepared	bigint	Number of prepared transaction commits
nabort_prepared	bigint	Number of prepared transaction aborts
nprovisional_waits	bigint	Number of update/delete operations on same tuples by concurrent apply transactions. These are provisional waits. See Parallel Apply
ntuple_waits	bigint	Number of update/delete operations that waited to be safely applied. See Parallel Apply
ncommit_waits	bigint	Number of fully applied transactions that had to wait before being committed. See Parallel Apply
stats_reset	timestamp with time zone	Time of the last statistics reset (performed by `bdr.reset_subscription_stats()`)

`bdr.stat_worker`

A view containing summary information and per worker statistics for PGD manager workers.

`bdr.stat_worker` columns

Column	Type	Description
worker_role	text	Role of the BDR worker
worker_pid	integer	Process id of the worker
sub_name	name	Name of the subscription the worker is related to, if any
worker_start	timestamp with time zone	Time at which the worker started
worker_xact_start	timestamp with time zone	Time at which the worker started the local db transaction, see `xact_start` in `pg_stat_activity` for more details
worker_xid	xid	Transaction id of the worker, see `backend_xid` in `pg_stat_activity` for more details
worker_xmin	xid	Oldest transaction id needed by the worker, see `backend_xmin` in `pg_stat_activity` for more details
worker_backend_state_change	timestamp with time zone	Backend state change timestamp see `state_change` in `pg_stat_activity` for more details
worker_backend_state	text	Current backend state see `state` in `pg_stat_activity` for more details
wait_event_type	text	The type of wait event the worker is currently waitiing on, if any (see `wait_event_type` in `pg_stat_activity` for more details)
wait_event	text	The exact event the worker is waiting on, if any (see `wait_event` in `pg_stat_activity` for more details)
blocked_by_pids	integer[]	List of PIDs blocking the worker, if any
query	text	Query currently being run by the worker

Column	Type	Description
worker_query_start	timestamp with time zone	Timestamp at which the current query run by the worker started

`bdr.stat_writer`

A view containing summary information and statistics for each subscription replication writer. There can be multiple writers for each subscription.

`bdr.stat_writer` columns

Column	Type	Description
worker_role	text	Role of the BDR worker (always 'writer')
worker_state	text	State of the worker (can be 'running', 'down', or 'disabled')
worker_pid	integer	Process id of the writer
sub_name	name	Name of the subscription the writer belongs to
writer_nr	integer	Writer index in the writer group for the same subscription
nxacts	bigint	The number of transactions the writer has processed since start
ncommits	bigint	The number of commits the writer processed since start
naborts	bigint	The number of aborts the writer processed since start
commit_queue_position	integer	Position in the commit queue, when serializing transactions against other writers in the same writer group
xact_source_xid	xid	Transaction id of the currently processed transaction on the source node
xact_source_commit_lsn	pg_lsn	LSN of the currently processed transaction on the source node
xact_nchanges	bigint	The number of changes in the currently processed transaction that have been written (updated every 1000 changes)
xact_origin_node_name	name	Origin node of the currently processed transaction
xact_origin_lsn	pg_lsn	Origin LSN of the currently processed transaction
xact_origin_timestamp	timestamp with time zone	Origin commit timestamp of the currently processed transaction
streaming_allowed	boolean	The writer can receive direct stream for large transactions
is_streaming	boolean	The writer is currently receiving a direct stream of a large transaction
nstream_file	bigint	The number of stream files the writer has processed
nstream_writer	bigint	The number of directly streamed transactions the writer has processed
worker_start	timestamp with time zone	The time at which the writer started
worker_xact_start	timestamp with time zone	The time at which the writer start the local db transaction (see xact_start in `pg_stat_activity` for more details)
worker_xid	xid	Transaction id of the worker (see `backend_xid` in `pg_stat_activity` for more details)
worker_xmin	xid	Oldest transaction id needed by the worker (see `backend_xmin` in `pg_stat_activity` for more details)
worker_backend_state_change	timestamp with time zone	Backend state change timestamp (see `state_change` in `pg_stat_activity` for more details)
worker_backend_state	text	Current backend state (see `state` in `pg_stat_activity` for more details)
wait_event_type	text	The type of wait event the writer is currently waiting on, if any (see `event_type` in `pg_stat_activity` for more details)
wait_event	text	The exact event the writer is waiting on, if any (see `wait_event` in `pg_stat_activity` for more details)

Column	Type	Description
blocked_by_pids	integer[]	List of PIDs blocking the writer, if any
query	text	Query currently being run by the writer (normally only set for DDL)
worker_query_start	timestamp with time zone	Timestamp at which the current query run by the worker started
command_progress_cmdtag	text	For commands with progress tracking, identifies the command current processed by the writer (can be one of 'CREATE INDEX', 'CREATE INDEX CONCURRENTLY', 'REINDEX', 'REINDEX CONCURRENTLY', 'CLUSTER', and 'VACUUM FULL')
command_progress_relation	text	For commands with progress tracking, identifies therelation which the command is working on
command_progress_phase	text	For commands with progress tracking, name of the current phase the command is in, refer to Progress Reporting in the Postgres documentation for details
command_progress_count	integer	For commands with progress tracking, the number of phases this command has gone through
command_progress_phase_nr	integer	For commands with progress tracking, the number of the phase of `command_progress_count`
command_progress_phase_tuples_total	real	For commands with progress tracking, the number of rows the current phase of the command has to process (if the phase is process rows)
command_progress_tuples_done	bigint	For commands with progress tracking, the number of rows the current phase of the command has already processed (if the phase is process rows)

`bdr.subscription`

This catalog table lists all the subscriptions owned by the local PGD node and their modes.

`bdr.subscription` columns

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

`bdr.subscription_summary`

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

`bdr.subscription_summary` columns

Name	Type	Description
node_group_name	name	Name of the PGD 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	OID of the PGD group the node is part of
sub_id	oid	OID of the subscription
origin_id	oid	OID of the origin node
target_id	oid	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.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	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.taskmgr_work_queue`

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

`bdr.taskmgr_work_queue` columns

Column	Type	Description
ap_wq_workid	bigint	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 the task belongs to
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_category	char	Work category; can be `c` (create partition), `m` (migrate partition), `d` (drop partition), or `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.taskmgr_workitem_status`

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

`bdr.taskmgr_workitem_status` columns

Column	Type	Description
ap_wi_workid	bigint	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	Status; can be `q` (queued), `c` (complete), `f` (failed), or `u` (unknown)
ap_wi_started_at	timestamptz	Start timestamptz of work item
ap_wi_finished_at	timestamptz	End timestamptz of work item

`bdr.taskmgr_local_work_queue`

Contains work items created and processed by the task manager. This is similar to `bdr.taskmgr_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.taskmgr_local_work_queue` columns

Column	Type	Description
ap_wq_workid	bigint	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 the task belongs to
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_category	char	Category; can be `c` (create partition), `m` (migrate partition), `d` (drop partition), or `a` (alter partition)
ap_wq_work_sql	text	SQL query for the work item

Column	Type	Description
ap_wq_work_depends	Oid[]	Always NULL

`bdr.taskmgr_local_workitem_status`

The status of the work items for locally managed tables.

`bdr.taskmgr_local_workitem_status` columns

Column	Type	Description
ap_wi_workid	bigint	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	Status; can be `q` (queued), `c` (complete), `f` (failed), or `u` (unknown)
ap_wi_started_at	timestamptz	Start timestamptz of work item
ap_wi_finished_at	timestamptz	End timestamptz of work item

`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	ID of the trigger
trigger_reloid	regclass	Name of the relating function
trigger_pgtgid	oid	Postgres trigger ID
trigger_type	char	Type of trigger call
trigger_name	name	Name of the trigger

`bdr.triggers`

An expanded view of `bdr.trigger` with columns that are easier to read.

Name	Type	Description
trigger_name	name	Name of the trigger
event_manipulation	text	Operations
trigger_type	bdr.trigger_type	Type of trigger
trigger_table	bdr.trigger_reloid	Table that calls the trigger
trigger_function	name	Function used

`bdr.workers`

Information about running PGD worker processes.

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

`bdr.writers`

Specific information about PGD 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
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
xact_nchanges	bigint	Number of changes processed by this writer (updated every 1000 rows)

`bdr.worker_tasks`

The `bdr.worker_tasks` view shows PGD'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 PGD node. State for all PGD 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

33.2 System functions

Perform PGD management primarily by using functions you call from SQL. All functions in PGD 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 version of the BDR extension currently in use.

`bdr.bdr_version_num`

This function retrieves the version number of the BDR extension 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

PGD exposes some parameters that you can query directly in SQL using, for example, `SHOW` or the `current_setting()` function. You can also use `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 Connection pools and proxies.

`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`

If a CAMO transaction is in progress, `transaction_id` is updated to show the assigned transaction id. You can query this parameter only by using using `PQparameterStatus` or equivalent. See Application use for a usage example.

`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 `timeout` for `TO ASYNC` 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 PGD 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 PGD group reconfiguration internal mechanisms.

`bdr.consensus_snapshot_export`

Synopsis

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

Generate a new PGD 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 PGD node group that's behind the exporting node's Raft log position.

The local PGD 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 PGD 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(snapshot
bytea)
```

Import a consensus snapshot that was exported by `bdr.consensus_snapshot_export()`, usually from another node in the same PGD 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 PGD 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 PGD 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(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(node_name text,
                             wait_for_completion boolean,
                             node_group_name text DEFAULT NULL)
```

Request the node identified by `node_name` to be the Raft leader. The request can be initiated from any of the PGD nodes and is internally forwarded to the current leader to transfer the leadership to the designated node. The designated node must be an ACTIVE PGD 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_global_lection_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.

The `node_group_name` is optional and can be used to specify the name of the node group where the leadership transfer happens. If not specified, it defaults to NULL, which is interpreted as the top-level group in the cluster. If the `node_group_name` is specified, the function transfers leadership only within the specified node group.

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 PGD 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 PGD 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 PGD node to wait for the local application of certain transactions originating from a given PGD 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, PGD 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.replicate_ddl_command`

Function to replicate a DDL command to a group of nodes.

Synopsis

```
bdr.replicate_ddl_command(ddl_cmd text,
                          replication_sets
text[],
                          ddl_locking
text,
                          execute_locally bool)
```

Parameters

- `ddl_cmd` — DDL command to execute.
- `replication_sets` — An array of replication set names to apply the `ddlcommand` to. If NULL (or the function is only passed the `ddlcommand`), this is set to the active PGD groups's default replication set.
- `ddl_locking` — A string that sets the `bdr.ddl_locking` value while replicating. Defaults to the GUC value for `bdr.ddl_locking` on the local system that's running `replicate_ddl_command`.
- `execute_locally` — A Boolean that determines whether the DDL command executes locally. Defaults to true.

Notes

The only required parameter of this function is `ddl_cmd`.

`bdr.replicate_ddl_command()` always replicates the command and is unaffected by the setting of `bdr.ddl_replication`.

`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` to avoid replication issues when the same replicated DDL command is sent to multiple nodes.

Be careful when using this function since you risk breaking replication and causing inconsistencies between nodes. For global schema changes, to replicate DDL, use either transparent DDL replication or `bdr.replicate_ddl_command()`.

`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` to avoid replication issues when the same replicated DDL command is sent to multiple nodes.

Be careful when using this function since you risk breaking replication and causing inconsistencies between nodes in the group. For global schema changes, to replicate DDL, use either transparent DDL replication or `bdr.replicate_ddl_command()` .

`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 `global` 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
-----------------
 global
```

`bdr.alter_node_kind`

PGD5 introduced a concept of Task Manager Leader node. The node is selected by PGD, but for upgraded clusters, it's important to set the `node_kind` properly for all nodes in the cluster. Do this manually after upgrading to the latest PGD version by calling the `bdr.alter_node_kind()` SQL function for each node.

Synopsis

```
bdr.alter_node_kind(node_name text,
                    node_kind
text);
```

Parameters

- `node_name` — Name of the node to change kind.
- `node_kind` — Kind of the node, which can be one of: `data`, `standby`, `witness`, or `subscriber-only`.

`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');
```

Global advisory locks

PGD supports global advisory locks. These locks are similar to the advisory locks available in PostgreSQL except that the advisory locks supported by PGD 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 released when the transaction ends unless you explicitly release it 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 to release it 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 PGD 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 PGD Raft information returned from the view `bdr.group_raft_details`.

Synopsis

```
bdr.monitor_group_raft()
```

Parameters

- `node_group_name` — The node group name to check.

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 PGD group slots.

This function also provides status information on subscriber-only nodes that are operating as subscriber-only group leaders in a PGD cluster whenoptimized topology is enabled.

Synopsis

```
bdr.monitor_local_replslots()
```

Notes

This function returns a record with fields `status` and `message`.

Status	Message
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

Further explaination is available in Monitoring replication slots.

`bdr.wal_sender_stats`

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

Synopsis

```
bdr.wal_sender_stats()
```

Output columns

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

`bdr.get_decoding_worker_stat`

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

Synopsis

```
bdr.get_decoding_worker_stat()
```

Output columns

- `pid` — The PID of the decoding worker (corresponds to the column `active_pid` in `pg_replication_slots`).
- `decoded_upto_lsn` — LSN up to which the decoding worker read transactional logs.
- `waiting` — Whether the decoding worker is waiting for new WAL.
- `waiting_for_lsn` — The LSN of the next expected WAL.

Notes

For 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_scope_id` — OID of the commit scope (see `bdr.commit_scopes`)).
- `sessions` — Number of sessions referencing the lag control entry.
- `current_commit_delay` — Current runtime commit delay, in fractional milliseconds.
- `maximum_commit_delay` — Configured maximum commit delay, in fractional milliseconds.
- `commit_delay_adjust` — Change to runtime commit delay possible during a sample interval, in fractional milliseconds.
- `curent_conforming_nodes` — Current runtime number of nodes conforming to lag measures.
- `minimum_conforming_nodes` — 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.
- `maximum_lag_bytes` — Configured maximum lag size, in kilobytes.
- `lag_time_threshold` — Lag time at which a commit delay is applied, in milliseconds.
- `maximum_lag_time` — Configured maximum lag time, in milliseconds.
- `sample_interval` — Configured minimum time between lag samples and possible commit delay adjustments, in milliseconds.

CAMO functions

CAMO requires that a client actively participates in the committing of a transaction by following the transactions progress. The functions listed here are used for that purpose and explained in CAMO.

`bdr.is_camo_partner_connected`

Allows checking of 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.

`bdr.is_camo_partner_ready`

Allows checking of 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, that is, before `timeout` for `TO ASYNC` 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.

`bdr.get_configured_camo_partner`

This function shows the local node's CAMO partner (configured by pair mode).

Synopsis

```
bdr.get_configured_camo_partner()
```

`bdr.wait_for_camo_partner_queue`

The function is a wrapper around `bdr.wait_for_apply_queue` defaulting to query the CAMO partner node. It returns an error if the local node isn't part of a CAMO pair.

Synopsis

```
bdr.wait_for_camo_partner_queue()
```

`bdr.camo_transactions_resolved`

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

Synopsis

```
bdr.camo_transactions_resolved()
```

`bdr.logical_transaction_status`

To check the status of a transaction that was being committed when the node failed, the application must use this function, passing as parameters the node id of the node the transaction originated from and the transaction id on the origin node.

Synopsis

```
bdr.logical_transaction_status(node_id OID, xid
OID,
                               require_camo_partner boolean DEFAULT true)
```

Parameters

- `node_id` — The node id of the PGD 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`.
- `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 wasn't a CAMO transaction.

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 is eventually replicated to all other PGD nodes. No need for the client to retry it.

- `'aborted'::TEXT` — The transaction was aborted and isn't replicated to any other PGD 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. Even transactions not yet replicated (or not even started on the origin node) might yield an `in progress` result on a peer PGD 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 because it's either in the future, not replicated to that specific node yet, or too far in the past. The status of such a transaction isn't yet or is 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.

Commit Scope functions

`bdr.add_commit_scope`

Deprecated. Use `bdr.create_commit_scope` instead. Previously, this function was used to add a commit scope to a node group. It's now deprecated and will emit a warning until it is removed in a future release, at which point it will raise an error.

`bdr.create_commit_scope`

`bdr.create_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.create_commit_scope(
    commit_scope_name NAME,
    origin_node_group NAME,
    rule TEXT,
    wait_for_ready boolean DEFAULT
true)
```

Note

`bdr.create_commit_scope` replaces the deprecated `bdr.add_commit_scope` function. Unlike `add_commit_scope`, it does not silently overwrite existing commit scopes when the same name is used. Instead, an error is reported.

`bdr.alter_commit_scope`

`bdr.alter_commit_scope` allows you to change a specific rule for a single origin node group in a commit scope.

Synopsis

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

`bdr.drop_commit_scope`

Drops 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.drop_commit_scope(
    commit_scope_name NAME,
    origin_node_group NAME)
```

> **Note**
>
> Dropping a commit scope that's still used as default by a node group isn't allowed.

`bdr.remove_commit_scope`

Deprecated. Use `bdr.drop_commit_scope` instead. Previously, this function was used to remove a commit scope from a node group. It's now deprecated and will emit a warning until it is removed in a future release, at which point it will raise an error.

33.3 PGD settings

You can set PGD-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 is `distributed`, which means `snowflakeid` is used for `int8` sequences (that is, `bigserial`) and `galloc` sequence for `int4` (that is, `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:

Value	Description
`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.
`auto`	Tables with PK are created with REPLICA IDENTITY DEFAULT, and tables without PK are created with REPLICA IDENTITY FULL. This is the default PGD behavior.

See the PostgreSQL documentation for more details.

PGD 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 included in the table definition, it won't be automatically picked up as `REPLICA IDENTITY`. You need to set the `REPLICA IDENTITY` explicitly using `ALTER TABLE ... REPLICA IDENTITY ...`.

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.

> **On setting bdr.default_replica_identity to default**
>
> When setting `bdr.default_replica_identity` to `default` using `ALTER SYSTEM`, always quote the value, like this:
>
> ```
> ALTER SYSTEM SET bdr.default_replica_identity="default";
> ```
>
> You need to include the quotes because default, unquoted, is a special value to the ALTER SYSTEM command that triggers the removal of the setting from the configuration file. When the setting is removed, the system uses the PGD default setting, which is `auto`.

`bdr.ddl_replication`

Automatically replicates DDL across nodes (default is `on`).

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

Running DDL or calling PGD 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 PGD replication functions is skipped due to this setting.

`bdr.role_replication`

Automatically replicates 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 parameter 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:

Value	Description
`all`	Use global locking for all DDL operations. (Default)
`dml`	Use global locking only for DDL operations that need to prevent writes by taking the global DML lock for a relation.
`off`	Don't use global locking for DDL operations.

Default is `all`.

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.

For backward compatibility, `bdr.ddl_locking` supports aliases. `on` and `true` are an alias for `all`. `false` is an alias for `off`.

See also Global locking.

`bdr.truncate_locking`

Sets the TRUNCATE command's locking behavior (default is `off`). When `true`, TRUNCATE obeys the `bdr.ddl_locking` setting.

Global locking

DDL locking is controlled by `bdr.ddl_locking`. Other global locking settings include the following.

`bdr.global_lock_max_locks`

Sets the maximum number of global locks that can be held on a node (default is 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 is 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 is 60 minutes). A value of zero disables this timeout.

`bdr.global_lock_idle_timeout`

Sets the maximum allowed duration of idle time in a transaction holding a global lock (default is 10 minutes). A value of zero disables this timeout.

`bdr.lock_table_locking`

Sets locking behavior for LOCK TABLE statement (default is off). When enabled, LOCK TABLE statement also takes a global DML lock on the cluster, blocking other locking statements.

Value	Description
`on`	Use global locking for all table locks.
`off`	Don't use global locking for table locks. (Default)

`bdr.predictive_checks`

Sets the 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, PGD checks that there are enough nodes connected and withing reasonable lag limit for getting the quorum needed by the global lock.

Node management

`bdr.replay_progress_frequency`

Sets the interval for sending replication position info to the rest of the cluster (default is 1 minute).

`bdr.standby_slot_names`

Sets the slots required 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`

Sets the default number of writers per subscription. (In PGD, 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 `bdr.writers_per_subscription` setting).

`bdr.xact_replication`

Replicates 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 PGD 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`

Overrides 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 that turns on batch processing of inserts for that table.

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

`bdr.maximum_clock_skew`

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 clocks on all servers are synchronized. The fact that the transaction is being replayed means it was committed in the past.

`bdr.maximum_clock_skew_action`

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

There are two possible values for this setting:

Value	Description
`WARN`	Log a warning about this fact. The warnings are logged once per minute 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`

Enables or disables connections to PGD (default is `on`).

Requires bdr_superuser or PostgreSQL superuser.

`bdr.standby_slot_names`

This setting is typically used in failover configurations to ensure that the failover-candidates streaming physical replicas for this PGD 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 WAL sender on a PGD node.

PGD's logical replication WAL senders ensure 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 PGD 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 doesn't move ahead of any of the regular PGD nodes. This can best be achieved by listing the logical slots of all regular PGD peer nodes in combination with setting `bdr.standby_slots_min_confirmed` to at least 1.

`bdr.standby_slots_min_confirmed`

Controls how many of the `bdr.standby_slot_names` have to confirm before sending data to PGD subscribers.

`bdr.writer_input_queue_size`

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`

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`

Allows for rate limiting of PGD 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 a 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's 1 minute.

If the backoff delay setting is changed and the PostgreSQL configuration is reloaded, then all current backoffs wait 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's 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, when 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.

Commit scope

`bdr.commit_scope`

Sets the current (or default) commit scope (default is an empty string).

Commit At Most Once

`bdr.camo_local_mode_delay`

The commit delay that applies in CAMO's asynchronous mode to emulate the overhead that normally occurs with the CAMO partner having to confirm transactions (default is 5 ms). Set to `0` to disable this feature.

`bdr.camo_enable_client_warnings`

Emits warnings if an activity is carried out in the database for which CAMO properties can't be guaranteed (default is enabled). Well-informed users can choose to disable this setting to reduce the amount of warnings going into their logs.

Transaction streaming

`bdr.default_streaming_mode`

Controls transaction streaming by the subscriber node. Possible 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

`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 PGD uses to write its debug messages. The default value is `debug2`. If you want to see detailed PGD debug output, set `bdr.debug_level = 'log'`.

`bdr.trace_level`

Similar to `bdr.debug_level`, defines the log level to use for PGD trace messages. Enabling tracing on all nodes of an EDB Postgres Distributed cluster might help EDB Support to diagnose issues. You can set this parameter 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. 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`

Tracks apply statistics for each subscription with `bdr.stat_subscription` (default is `on`).

`bdr.track_relation_apply`

Tracks apply statistics for each relation with `bdr.stat_relation` (default is `off`).

`bdr.track_apply_lock_timing`

Tracks lock timing when tracking statistics for relations with `bdr.stat_relation` (default is `off`).

Decoding worker

`bdr.enable_wal_decoder`

Enables logical change record (LCR) sending on a single node with a decoding worker (default is false). When set to true, a decoding worker process starts, and WAL senders send the LCRs it produces. If set back to false, any WAL senders using LCR are restarted and use the WAL directly.

> **Note**
>
> You also need to enable this setting on all nodes in the PGD group and set the `enable_wal_decoder` option to true on the group.

`bdr.receive_lcr`

When subscribing to another node, this setting enables the node to request the use of logical change records (LCRs) for the subscription (default is false). When this setting is true on a downstream node, the node requests that upstream nodes use LCRs when sending to it. If you set `bdr.enable_wal_decoder` to true on a node, also set this setting to true.

> **Note**
>
> You also need to enable this setting on all nodes in the PGD group and set the `enable_wal_decoder` option to true on the group.

`bdr.lcr_cleanup_interval`

Logical change record (LCR) file cleanup interval (default is 3 minutes). When the decoding worker is enabled, the decoding worker stores LCR files as a buffer. These files are periodically cleaned, and this setting controls the interval between any two consecutive cleanups. Setting it to zero disables cleanup.

Connectivity settings

The following are a set of connectivity settings affecting all cross-node `libpq` connections. The defaults are set to fairly conservative values and cover most production needs. All variables have `SIGHUP` context, meaning changes are applied upon reload.

`bdr.global_connection_timeout`

Maximum time to wait while connecting, in seconds (default is 15 seconds). Write as a decimal integer, for example, 10. Zero, negative, or not specified means wait indefinitely. The minimum allowed timeout is 2 seconds, therefore a value of 1 is interpreted as 2.

`bdr.global_keepalives`

Controls whether TCP keepalives are used (default is 1, meaning on). If you don't want keepalives, you can change this to 0, meaning off. This parameter is ignored for connections made by a Unix-domain socket.

`bdr.global_keepalives_idle`

Controls the number of seconds of inactivity after which TCP sends a keepalive message to the server (default is 1 second). A value of zero uses the system default. This parameter is ignored for connections made by a Unix-domain socket or if keepalives are disabled. It's supported only on systems where `TCP_KEEPIDLE` or an equivalent socket option is available. On other systems, it has no effect.

`bdr.global_keepalives_interval`

Controls the number of seconds after which to retransmit a TCP keepalive message that isn't acknowledged by the server (default is 2 seconds). A value of zero uses the system default. This parameter is ignored for connections made by a Unix-domain socket or if keepalives are disabled. It's supported only on systems where `TCP_KEEPINTVL` or an equivalent socket option is available. On other systems, it has no effect.

`bdr.global_keepalives_count`

Controls the number of TCP keepalives that can be lost before the client's connection to the server is considered dead (default is 3). A value of zero uses the system default. This parameter is ignored for connections made by a Unix-domain socket or if keepalives are disabled. It's supported only on systems where `TCP_KEEPCNT` or an equivalent socket option is available. On other systems, it has no effect.

`bdr.global_tcp_user_timeout`

Controls the number of milliseconds that transmitted data can remain unacknowledged before a connection is forcibly closed (default is 5000, that is, 5 seconds). A value of zero uses the system default. This parameter is ignored for connections made by a Unix-domain socket. It's supported only on systems where `TCP_USER_TIMEOUT` is available. On other systems, it has no effect.

Topology settings

`bdr.force_full_mesh`

Forces the full mesh topology (default is `on`). When set to `off`, PGD will attempt to use the optimized topology for subscriber-only groups. This setting is only effective when the requirements for the optimized topology are met. See Optimizing subscriber-only groups for more information.

Internal settings - Raft timeouts

`bdr.raft_global_election_timeout`

To account for network failures, the Raft consensus protocol implements timeouts for elections and requests. This value is used when a request is being sent to the global (top-level) group. The default is 6 seconds (6s).

`bdr.raft_group_election_timeout`

To account for network failures, the Raft consensus protocol implements timeouts for elections and requests. This value is used when a request is being sent to the sub-group. The default is 3 seconds (3s).

`bdr.raft_response_timeout`

For responses, the settings of `bdr.raft_global_election_timeout` and `bdr.raft_group_election_timeout` are used as appropriate. You can override this behavior by setting this variable. The setting of `bdr.raft_response_timeout` must be less than either of the election timeout values. Set this variable to -1 to disable the override. The default is -1.

Internal settings - Other Raft values

`bdr.raft_keep_min_entries`

The minimum number of entries to keep in the Raft log when doing log compaction (default is `1000`; PGD 5.3 and earlier: `100`). The value of `0` disables log compaction. You can set this parameter only at Postgres server start.

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

`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 is 3000 ms.

`bdr.raft_log_min_message_duration`

When to log a consensus request. Measures roundtrip time of a PGD consensus request and logs an `INFO` message if the time exceeds this parameter (default is 5000 ms).

`bdr.raft_group_max_connections`

The maximum number of connections across all PGD groups for a Postgres server (default is 100 connections). These connections carry PGD consensus requests between the groups' nodes. You can set this parameter only at Postgres server start.

Internal settings - Other values

`bdr.backwards_compatibility`

Specifies the version to be backward compatible to, in the same numerical format as used by `bdr.bdr_version_num`, for example, `30618`. (Default is the current PGD version.) Enables exact behavior of a former PGD version, even if this has generally unwanted effects. 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`

Tracks 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`

PGD monitors 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 is 0.1.

`bdr.enable_auto_sync_reconcile`

When enabled, nodes perform automatic synchronization of data from a node that is furthest ahead with respect to the down node. Default (from 5.5.1) is off.

33.4 Node management

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 `bdr.promote_node` was just called 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 PGD node is in progress.
- `ACTIVE` : The node is a full PGD node and is currently `ACTIVE`. This is the most common node status.
- `PART_START` : Node was `ACTIVE` or `STANDBY` and `bdr.part_node` was just called 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 commands

PGD also provides a command-line utility for adding nodes to the PGD group using a physical copy (`pg_basebackup`) of an existing node.

`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 `pg_basebackup -X stream` to fill the directory using a fast block-level copy operation.

If the specified data directory isn't empty, it's used as the base for the new node. Initially, it waits for catchup and then promotes to a master node before joining the PGD group. The `--standby` option, if used, turns it into a logical standby node.

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 PGD group. This is true whether `bdr_init_physical` makes its own base backup or an existing base backup is being promoted to a new PGD node. Thus the `--replication-sets` option affects only the data published and subscribed to after the node joins the PGD 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.

33.5 Node management interfaces

You can add and remove nodes dynamically using the SQL interfaces.

`bdr.alter_node_group_option`

Modifies a PGD node group configuration.

Synopsis

```
bdr.alter_node_group_option(node_group_name text,
    config_key text,
    config_value text)
```

Parameters

Name	Description
`node_group_name`	Name of the group to change.
`config_key`	Key of the option in the node group to change.
`config_value`	New value to set for the given key.

`config_value` is parsed into the data type appropriate for the option.

The table shows the group options that can be changed using this function.

Name	Type	Description
`apply_delay`	`interval`	How long nodes wait to apply incoming changes. This option is useful mainly to set up a special subgroup with delayed subscriber-only nodes. Don't set this on groups that contain data nodes or on the top-level group. Default is `0s`.
`check_constraints`	`boolean`	Whether the apply process checks the constraints when writing replicated data. We recommend keeping the default value or you risk data loss. Valid values are `on` or `off`. Default is `on`.
`default_commit_scope`	`text`	The commit scope to use by default, initially the `local` commit scope. This option 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.
`enable_proxy_routing`	`boolean`	Where `pgd-proxy` through the group leader is enabled for given group. Valid values are `on` or `off`. Default is `off`.
`enable_raft`	`boolean`	Whether group has its own Raft consensus. This option is necessary for setting `enable_proxy_routing` to `on`. This option is always `on` for the top-level group. Valid values are `on` or `off`. Default is `off` for subgroups.

Name	Type	Description
`enable_wal_decoder`	`boolean`	Enables/disables the decoding worker process. You can't enable the decoding worker process if `streaming_mode` is already enabled. Valid values are `on` or `off`. Default is `off`.
`location`	`text`	Information about group location. This option is purely metadata for monitoring. Default is `''` (empty string).
`num_writers`	`integer`	Number of parallel writers for the subscription backing this node group. Valid values are `-1` or a positive integer. `-1` means the value specified by the GUC `bdr.writers_per_subscription` is used. `-1` is the default.
`route_reader_max_lag`	`integer`	Maximum lag in bytes for a node to be considered a viable read-only node. Currently reserved for future use.
`route_writer_max_lag`	`integer`	Maximum lag in bytes of the new write candidate to be selected as write leader. If no candidate passes this, no writer is selected. Default is `-1`.
`route_writer_wait_flush`	`boolean`	Whether to switch if PGD needs to wait for the flush. Currently reserved for future use.
`streaming_mode`	`text`	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 transactions, then the changes are written to a file and applied after the transaction is committed. If the value is set to `auto`, PGD 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. Default is `auto`. For more details, see Transaction streaming.

Return value

`bdr.alter_node_group_option()` returns `VOID` on success.

An `ERROR` is raised if any of the provided parameters is invalid.

Notes

You can examine the current state of node group options by way of the view `bdr.node_group_summary`.

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.

bdr.alter_node_interface

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

Synopsis

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

Parameters

Name	Description
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 PGD group, including the node that's 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_node_option

Modifies the PGD node routing configuration.

Synopsis

```
bdr.alter_node_option(node_name text,
    config_key text,
    config_value
text);
```

Parameters

Name	Description
node_name	Name of the node to change.
config_key	Key of the option in the node to change.
config_value	New value to set for the given key.

The node options you can change using this function are:

Config Key	Description
`route_priority`	Relative routing priority of the node against other nodes in the same node group. Default is `'-1'` .
`route_fence`	Whether the node is fenced from routing. When true, the node can't receive connections from PGD Proxy. Default is `'f'` (false).
`route_writes`	Whether writes can be routed to this node, that is, whether the node can become write leader. Default is `'t'` (true) for data nodes and `'f'` (false) for other node types.
`route_reads`	Whether read-only connections can be routed to this node. Currently reserved for future use. Default is `'t'` (true) for data and subscriber-only nodes, `'f'` (false) for witness and standby nodes.
`route_dsn`	The dsn for the proxy to use to connect to this node. This option is optional. If not set, it defaults to the node's `node_dsn` value.

`bdr.alter_subscription_enable`

Enables either the specified subscription or all the subscriptions of the local PGD 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

Name	Description
`subscription_name`	Name of the subscription to enable. If NULL (the default), all subscriptions on the local node are enabled.
`immediate`	Used to force the action immediately, starting all the workers associated with the enabled subscription. When this option is `true` , you can't run this function inside of the transaction block.

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`

Disables either the specified subscription or all the subscriptions of the local PGD 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

Name	Description
`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.

`bdr.create_node`

Creates a node.

Synopsis

```
bdr.create_node(node_name text,
                local_dsn text,
                node_kind DEFAULT NULL)
```

Parameters

Name	Description
`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.
`node_kind`	One of `data` (the default), `standby`, `subscriber-only`, or `witness`. If you don't set this parameter, or if you provide `NULL`, the default `data` node kind is used.

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 node until the end of the transaction.

`bdr.create_node_group`

Creates a PGD node group. By default, the local node joins the group as the only member. You can add more nodes to the group with `bdr.join_node_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

Name	Description
`node_group_name`	Name of the new PGD group. As with the node name, valid group names consist of only lowercase letters, numbers, and underscores.
`parent_group_name`	If a node subgroup is being created, this must be the name of the parent group. Provide `NULL` (the default) when creating the main node group for the cluster.
`join_node_group`	Determines whether the node joins the group being created. The default value is `true`. Providing `false` when creating a subgroup means the local node won't join the new group, for example, when creating an independent remote group. In this case, you must specify `parent_group_name`.
`node_group_type`	The valid values are `NULL` or `subscriber-only`. `NULL` (the default) is for creating a normal, general-purpose node group. `subscriber-only` is for creating subscriber-only groups whose members receive changes only from the fully joined nodes in the cluster but that never send changes to other nodes.

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.drop_node_group

Drops an empty PGD node group. If there are any joined nodes in the group, the function will fail.

Synopsis

```
bdr.drop_node_group(node_group_name text)
```

Parameters

Name	Description
node_group_name	Name of the PGD group to drop.

Notes

This function passes a request to the group consensus mechanism to drop the group. The function isn't transactional. The dropping process happens in the background, and you can't roll it back.

bdr.join_node_group

Joins the local node to an already existing PGD group.

Synopsis

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

Parameters

Name	Description
join_target_dsn	Specifies the connection string to an existing (source) node in the PGD group you want to add the local node to.
node_group_name	Optional name of the PGD group. Defaults to NULL, which tries to detect the group name from information present on the source node.

Name	Description
`wait_for_completion`	Wait for the join process to complete before returning. Defaults to `true`.
`synchronize_structure`	Specifies whether to perform database structure (schema) synchronization during the join. `all`, the default setting, synchronizes the complete database structure. `none` does not synchronize any structure. However, data will still be synchronized, meaning the database structure must already be present on the joining node. Note that by design, neither schema nor data will ever be synchronized to witness nodes.
`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. This option is deprecated and will be disabled or removed in future versions of PGD.

> **Warning**
>
> `pause_in_standby` is deprecated since BDR 5.0. The recommended way to create a logical standby is to set `node_kind` to `standby` when creating the node with `bdr.create_node`.

If `wait_for_completion` is specified as `false`, the function call returns as soon as the joining procedure starts. You can see the progress of the join in the log files and the `bdr.event_summary` information view. You can call the function `bdr.wait_for_join_completion()` after `bdr.join_node_group()` to wait for the join operation to complete. It can emit progress information if called with `verbose_progress` set to `true`.

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 PGD group.

`bdr.part_node`

Removes (parts) the node from the PGD group but doesn't remove data from the node.

You can call the function from any active node in the PGD 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, use `bdr.alter_subscription_disable()`.

Synopsis

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

Parameters

Name	Description
`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 PGD 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's 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.promote_node`

Promotes a local logical standby node to a full member of the PGD 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.switch_node_group`

Switches the local node from its current subgroup to another subgroup in the same existing PGD node group.

Synopsis

```
bdr.switch_node_group
(
    node_group_name text,
    wait_for_completion boolean DEFAULT
true
)
```

Parameters

Name	Description
`node_group_name`	Name of the PGD group or subgroup.
`wait_for_completion`	Wait for the switch process to complete before returning. Defaults to `true`.

If `wait_for_completion` is set to `false`, this is an asynchronous call that returns as soon as the switching procedure starts. You can see progress of the switch in logs and the `bdr.event_summary` information view or by calling the `bdr.wait_for_join_completion()` function after `bdr.switch_node_group()` returns.

Notes

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

The function isn't transactional. The switching 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 local node changes membership from its current subgroup to another subgroup in the same PGD node group without needing to part the cluster. The node's kind must match that of existing nodes in the target subgroup.

Node switching doesn't hold any locks in the PGD group.

Restrictions: currently, the function allows switching only between a subgroup and its PGD node group. To effect a move between subgroups you need to make two separate calls: 1) switch from subgroup to node group and, 2) switch from node group to other subgroup.

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

Name	Description
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.alter_node_group_config

Changes the configuration parameters of an existing PGD group. Options with NULL value (default for all of them) aren't modified.

> **Warning**
>
> This function exists only for compatibility with PGD4 and 3.7. Use `bdr.alter_node_group_option` instead.

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

Name	Description
`node_group_name`	Name of an existing PGD group. The local node must be part of the group.
`insert_to_update`	Reserved for backward compatibility.
`update_to_insert`	Reserved for backward compatibility. This option is deprecated and will be disabled or removed in future versions of PGD. Use `bdr.alter_node_set_conflict_resolver` instead.
`ignore_redundant_updates`	Reserved for backward compatibility.
`check_full_tuple`	Reserved for backward compatibility.
`apply_delay`	How long nodes wait to apply incoming changes. This parameter is useful mainly to set up a special subgroup with delayed subscriber-only nodes. Don't set this on groups that contain data nodes or on the top-level group. Default is `0s`.
`check_constraints`	Whether the apply process checks the constraints when writing replicated data. We recommend keeping this set to the default value or you risk data loss. Valid values are `on` or `off`. Default is `on`. Applies to top-level group only.
`num_writers`	Number of parallel writers for the subscription backing this node group. Valid values are `-1` or a positive integer. `-1` means the value specified by the GUC `bdr.writers_per_subscription` is used. `-1` is the default. Applies to top-level group only.
`enable_wal_decoder`	Enables/disables the decoding worker process. You can't enable the decoding worker process if `streaming_mode` is already enabled. Valid values are `on` or `off`. Default is `off`. Applies to top-level group only.
`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`, PGD 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. Applies to top-level group only.

Name	Description
`default_commit_scope`	The commit scope to use by default, initially the `local` commit scope. This parameter 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.

33.6 Routing functions

`bdr.create_proxy`

Create a proxy configuration.

Synopsis

```
bdr.create_proxy(proxy_name text, node_group text, proxy_mode
text);
```

Parameters

Name	Type	Default	Description
`proxy_name`	text		Name of the new proxy.
`node_group`	text		Name of the group to be used by the proxy.
`proxy_mode`	text	`'default'`	Mode of the proxy. It can be `'default'` (listen_port connections follow write leader, no read_listen_port), `'read-only'` (no listen_port, read_listen_port connections follow read-only nodes), or `'any'` (listen_port connections follow write_leader, read_listen_port connections follow read-only nodes). Default is `'default'`.

When proxy_mode is set to `'default'`, all read options in the proxy config are set to NULL. When it's set to `'read-only'`, all write options in the proxy config are set to NULL. When set to `'any'` all options are set to their defaults.

`bdr.alter_proxy_option`

Change a proxy configuration.

Synopsis

```
bdr.alter_proxy_option(proxy_name text, config_key text, config_value
text);
```

Parameters

Name	Type	Default	Description
`proxy_name`	text		Name of the proxy to change.
`config_key`	text		Key of the option in the proxy to change.

Name	Type	Default	Description
config_value	text		New value to set for the given key.

The table shows the proxy options (config_key) that can be changed using this function.

Option	Description
listen_address	Address for the proxy to listen on. Default is '{0.0.0.0}'.
listen_port	Port for the proxy to listen on. Default is '6432' in 'default' or 'any' mode and '0' in 'read-only' mode, which disables the write leader following port.
max_client_conn	Maximum number of connections for the proxy to accept. Default is '32767'.
max_server_conn	Maximum number of connections the proxy can make to the Postgres node. Default is '32767'.
server_conn_timeout	Connection timeout for server connections. Default is '2' (seconds).
server_conn_keepalive	Keepalive interval for server connections. Default is '10' (seconds).
consensus_grace_period	Duration for which proxy continues to route even upon loss of a Raft leader. If set to 0s, proxy stops routing immediately. Default is generally '6' (seconds) for local proxies and '12' (seconds) for global proxies. These values will be overridden if raft_response_timeout , raft_global_election_timeout , or raft_group_election_timeout are changed from their defaults.
read_listen_address	Address for the read-only proxy to listen on. Default is '{0.0.0.0}'.
read_listen_port	Port for the read-only proxy to listen on. Default is '6433' in 'read-only' or 'any' mode and '0' in 'default' mode, which disables the read-only port.
read_max_client_conn	Maximum number of connections for the read-only proxy to accept. Default is '32767'.
read_max_server_conn	Maximum number of connections the read-only proxy can make to the Postgres node. Default is '32767'.
read_server_conn_keepalive	Keepalive interval for read-only server connections. Default is '10' (seconds).
read_server_conn_timeout	Connection timeout for read-only server connections. Default is '2' (seconds).
read_consensus_grace_period	Duration for which read-only proxy continues to route even upon loss of a Raft leader. Default is 1 hour.

Changing any of these values requires a restart of the proxy.

`bdr.drop_proxy`

Drop a proxy configuration.

Synopsis

```
bdr.drop_proxy(proxy_name
text);
```

Parameters

Name	Type	Default	Description
`proxy_name`	text		Name of the proxy to drop.

`bdr.routing_leadership_transfer`

Changing the routing leader transfers the leadership of the node group to another node.

Synopsis

```
bdr.routing_leadership_transfer(node_group_name text,
             leader_name
text,
             transfer_method text DEFAULT 'strict',
             transfer_timeout interval DEFAULT
'10s');
```

Parameters

Name	Type	Default	Description
`node_group_name`	text		Name of group where the leadership transfer is requested.
`leader_name`	text		Name of node that will become write leader.
`transfer_method`	text	`'strict'`	Type of the transfer. It can be `'fast'` or the default, `'strict'`, which checks the maximum lag.
`transfer_timeout`	interval	'10s'	Timeout of the leadership transfer. Default is 10 seconds.

33.7 Commit scopes

Commit scopes are rules that determine how transaction commits and conflicts are handled within a PGD system. You can read more about them inCommit Scopes.

You can manipulate commit scopes using the following functions:

- `bdr.create_commit_scope`
- `bdr.alter_commit_scope`
- `bdr.drop_commit_scope`

Commit scope syntax

The overall grammar for commit scope rules is composed as follows:

```
commit_scope:
    commit_scope_operation [AND ...]

commit_scope_operation:
    commit_scope_group confirmation_level commit_scope_kind

commit_scope_target:
    { (node_group [, ...])
      | ORIGIN_GROUP }

commit_scope_group:
{ ANY num [NOT] commit_scope_target
  | MAJORITY [NOT] commit_scope_target
  | ALL [NOT] commit_scope_target }

confirmation_level:
    [ ON { received|replicated|durable|visible } ]

commit_scope_kind:
{ GROUP COMMIT [ ( group_commit_parameter = value [, ... ] ) ] [ ABORT ON ( abort_on_parameter = value ) ] [
DEGRADE ON (degrade_on_parameter = value [, ... ] ) TO commit_scope_degrade_operation ]
  | CAMO [ DEGRADE ON ( degrade_on_parameter = value [, ... ] ) TO ASYNC ]
  | LAG CONTROL [ ( lag_control_parameter = value [, ... ] ) ]
  | SYNCHRONOUS_COMMIT [ DEGRADE ON (degrade_on_parameter = value ) TO commit_scope_degrade_operation ] }

commit_scope_degrade_operation:
   commit_scope_group confirmation_level commit_scope_kind
```

Where `node_group` is the name of a PGD data node group.

commit_scope_degrade_operation

The `commit_scope_degrade_operation` is either the same commit scope kind with a less restrictive commit scope group as the overall rule being defined, or is asynchronous (`ASYNC`).

For instance, you can degrade from an `ALL SYNCHRONOUS_COMMIT` to a `MAJORITY SYNCHRONOUS_COMMIT` or a `MAJORITY SYNCHRONOUS_COMMIT` to an `ANY 3 SYNCHRONOUS_COMMIT` or even an `ANY 3 SYNCHRONOUS_COMMIT` to an `ANY 2 SYNCHRONOUS_COMMIT` . You can also degrade from `SYNCRHONOUS_COMMIT` to `ASYNC` . However, you cannot degrade from `SYNCHRONOUS_COMMIT` to `GROUP_COMMIT` or the other way around, regardless of the commit scope groups involved.

It is also possible to combine rules using `AND` , each with their own degradation clause:

```
ALL ORIGIN_GROUP SYNCHRONOUS_COMMIT DEGRADE ON (timeout = 10s) TO MAJORITY ORIGIN_GROUP SYNCHRONOUS COMMIT AND
ANY 1 NOT ORIGIN_GROUP SYNCHRONOUS_COMMIT DEGRADE ON (timeout = 20s) TO ASYNC
```

Commit scope targets

ORIGIN_GROUP

Instead of targeting a specific group, you can also use `ORIGIN_GROUP`, which dynamically refers to the bottommost group from which a transaction originates. Therefore, if you have a top level group, `top_group`, and two subgroups as children, `left_dc` and `right_dc`, then adding a commit scope like:

```
SELECT
bdr.create_commit_scope(
    commit_scope_name := 'example_scope',
    origin_node_group := 'top_level_group',
    rule := 'MAJORITY ORIGIN_GROUP
SYNCHRONOUS_COMMIT',
    wait_for_ready :=
true
);
```

would mean that for transactions originating on a node in `left_dc`, a majority of the nodes of `left_dc` would need to confirm the transaction synchronously before the transaction is committed. Moreover, the same rule would also mean that for transactions originating from a node in `right_dc`, a majority of nodes from `right_dc` are required to confirm the transaction synchronously before it is committed. This saves the need to add two seperate rules, one for `left_dc` and one for `right_dc`, to the commit scope.

Commit scope groups

ANY

Example: `ANY 2 (left_dc)`

A transaction under this commit scope group will be considered committed after any two nodes in the `left_dc` group confirm they processed the transaction.

ANY NOT

Example: `ANY 2 NOT (left_dc)`

A transaction under this commit scope group will be considered committed if any two nodes that aren't in the `left_dc` group confirm they processed the transaction.

MAJORITY

Example: `MAJORITY (left_dc)`

A transaction under this commit scope group will be considered committed if a majority of the nodes in the `left_dc` group confirm they processed the transaction.

MAJORITY NOT

Example: `MAJORITY NOT (left_dc)`

A transaction under this commit scope group will be considered committed if a majority of the nodes that aren't in the `left_dc` group confirm they processed the transaction.

ALL

Example: `ALL (left_dc)`

A transaction under this commit scope group will be considered committed if all of the nodes in the `left_dc` group confirm they processed the transaction.

When `ALL` is used with `GROUP COMMIT`, the `commit_decision` setting must be set to `raft` to avoid reconciliation issues.

ALL NOT

Example: `ALL NOT (left_dc)`

A transaction under this commit scope group will be considered committed if all of the nodes that aren't in the `left_dc` group confirm they processed the transaction.

Confirmation level

The confirmation level sets the point in time when a remote PGD node confirms that it reached a particular point in processing a transaction.

ON received

A transaction is confirmed immediately after receiving it, prior to starting the local application.

ON replicated

A transaction is confirmed after applying changes of the transaction but before flushing them to disk.

ON durable

A transaction is confirmed after all of its changes are flushed to disk.

ON visible

This is the default visibility. A transaction is confirmed after all of its changes are flushed to disk and it's visible to concurrent transactions.

Commit Scope kinds

More details of the commit scope kinds and details of their parameters:

- Synchronous Commit
- Group Commit
- CAMO (Commit At Most Once)
- Lag Control

> **Parameter values**
>
> Specify Boolean, enum, int, and interval values using the Postgres GUC parameter value conventions.

SYNCHRONOUS_COMMIT

```
SYNCHRONOUS_COMMIT [ DEGRADE ON (degrade_on_parameter = value ) TO commit_scope_degrade_operation ]
```

DEGRADE ON parameters

Parameter	Type	Default	Description
`timeout`	interval	0	Timeout in milliseconds (accepts other units) after which operation degrades. (0 means not set.)
`require_write_lead`	Boolean	False	Specifies whether the node must be a write lead to be able to switch to degraded operation.

These set the conditions on which the commit scope rule will degrade to a less restrictive mode of operation.

commit_scope_degrade_operation

The `commit_scope_degrade_operation` must be `SYNCHRONOUS_COMMIT` with a less restrictive commit scope group—or must be asynchronous (`ASYNC`).

GROUP COMMIT

Allows commits to be confirmed by a consensus of nodes, controls conflict resolution settings, and, like `SYNCHRONOUS_COMMIT` , has optional rule-degredation parameters.

```
GROUP COMMIT [ ( group_commit_parameter = value [, ...] ) ] [ ABORT ON ( abort_on_parameter = value ) ] [ DEGRADE ON (degrade_on_parameter = value ) TO commit_scope_degrade_operation ]
```

GROUP COMMIT parameters

Parameter	Type	Default	Description
`transaction_tracking`	Boolean	Off/False	Specifies whether to track status of transaction. See transaction_tracking settings.
`conflict_resolution`	enum	async	Specifies how to handle conflicts. (`async` | `eager`). See conflict_resolution settings.

Parameter	Type	Default	Description
`commit_decision`	enum	group	Specifies how the COMMIT decision is made. (`group` | `partner` | `raft`). See commit_decision settings.

ABORT ON parameters

Parameter	Type	Default	Description
`timeout`	interval	0	Timeout in milliseconds (accepts other units). (0 means not set.)
`require_write_lead`	Boolean	False	CAMO only. If set, then for a transaction to switch to local (async) mode, a consensus request is required.

DEGRADE ON parameters

Parameter	Type	Default	Description
`timeout`	interval	0	Timeout in milliseconds (accepts other units) after which operation degrades. (0 means not set.)
`require_write_lead`	Boolean	False	Specifies whether the node must be a write lead to be able to switch to degraded operation.

transaction_tracking settings

When set to true, two-phase commit transactions:

- Look up commit decisions when a writer is processing a PREPARE message.
- When recovering from an interruption, look up the transactions prepared before the interruption. When found, it then looks up the commit scope of the transaction and any corresponding RAFT commit decision. Suppose the node is the origin of the transaction and doesn't have a RAFT commit decision, and `transaction_tracking` is on in the commit scope. In that case, it periodically looks for a RAFT commit decision for this unresolved transaction until it's committed or aborted.

conflict_resolution settings

The value `async` means resolve conflicts asynchronously during replication using the conflict resolution policy.

The value `eager` means that conflicts are resolved eagerly during COMMIT by aborting one of the conflicting transactions.

Eager is only available with `MAJORITY` or `ALL` commit scope groups.

When used with the `ALL` commit scope group, the `commit_decision` must be set to `raft` to avoid reconcilation issue.

See "Conflict resolution" in Group Commit.

commit_decision settings

The value `group` means the preceding `commit_scope_group` specification also affects the COMMIT decision, not just durability.

The value `partner` means the partner node decides whether transactions can be committed. This value is allowed only on groups with 2 data nodes.

The value `raft` means the decision makes use of PGD's built-in Raft consensus. Once all the nodes in the selected commit scope group have confirmed the transaction, to ensure that all the nodes in the PGD cluster have noted the transaction, it is noted with the all-node Raft.

This option must be used when the `ALL` commit scope group is being used to ensure no divergence between the nodes over the decision. This option may have low performance.

See "Commit decisions" in Group Commit.

commit_scope_degrade_operation settings

The `commit_scope_degrade_operation` must be `GROUP_COMMIT` with a less restrictive commit scope group—or must be asynchronous (`ASYNC`).

CAMO

With the client's cooperation, enables protection to prevent multiple insertions of the same transaction in failover scenarios.

See "CAMO" in Durability for more details.

```
CAMO [ DEGRADE ON ( degrade_on_parameter = value ) TO ASYNC ]
```

DEGRADE ON parameters

Allows degrading to asynchronous operation on timeout.

Parameter	Type	Default	Description
`timeout`	interval	0	Timeout in milliseconds (accepts other units) after which operation becomes asynchronous. (0 means not set.)
`require_write_lead`	Boolean	False	Specifies whether the node must be a write lead to be able to switch to asynchronous mode.

LAG CONTROL

Allows the configuration of dynamic rate-limiting controlled by replication lag.

See "Lag Control" in Durability for more details.

```
LAG CONTROL [ ( lag_control_parameter = value [, ... ] ) ]
```

LAG CONTROL parameters

Parameter	Type	Default	Description
`max_lag_size`	int	0	The maximum lag in kB that a given node can have in the replication connection to another node. When the lag exceeds this maximum scaled by `max_commit_delay`, lag control adjusts the commit delay.
`max_lag_time`	interval	0	The maximum replication lag in milliseconds that the given origin can have with regard to a replication connection to a given downstream node.

Parameter	Type	Default	Description
`max_commit_delay`	interval	0	Configures the maximum delay each commit can take, in fractional milliseconds. If set to 0, it disables Laq Control. After each commit delay adjustment (for example, if the replication is lagging more than `max_lag_size` or `max_lag_time`), the commit delay is recalculated with the weight of the `bdr.lag_control_commit_delay_adjust` GUC. The `max_commit_delay` is a ceiling for the commit delay.

- If `max_lag_size` and `max_lag_time` are set to 0, the LAG CONTROL is disabled.
- If `max_commit_delay` is not set or set to 0, the LAG CONTROL is disabled.

The lag size is derived from the delta of the `send_ptr` of the walsender to the `apply_ptr` of the receiver.

The lag time is calculated according to the following formula:

```
lag_time = (lag_size / apply_rate) * 1000;
```

Where `lag_size` is the delta between the `send_ptr` and `apply_ptr` (as used for `max_lag_size`), and `apply_rate` is a weighted exponential moving average, following the simplified formula:

```
apply_rate = prev_apply_rate * (1 - apply_rate_weight) +
             ((apply_ptr_diff * apply_rate_weight) / diff_secs);
```

Where:

- `prev_apply_rate` was the previously configured `apply_rate`, before recalculating the new rate.
- `apply_rate_weight` is the value of the GUC `bdr.lag_tracker_apply_rate_weight`.
- `apply_ptr_diff` is the difference between the current `apply_ptr` and the `apply_ptr` at the point in time when the apply rate was last computed.
- `diff_secs` is the delta in seconds from the last time the apply rate was calculated.

33.8 Conflicts

Conflict detection

List of conflict types

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

Conflict type	Description
`insert_exists`	An incoming insert conflicts with an existing row by way of 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. PGD 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`.

PGD lets you override the default behavior of conflict resolution by using the following function.

List of conflict resolvers

Several conflict resolvers are available in PGD, with differing coverages of the conflict types they can handle:

Resolver	Description
`error`	Throws an error and stops replication.

Resolver	Description
`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`	Skips 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. This 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 recommend that when using this resolver, you don't reuse the object names immediately after they're dropped.
`skip_transaction`	Skips the whole transaction that generated the conflict.
`update_if_newer`	Updates 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 performs 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`	Tries 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, skips the change. Can be used for `update_missing` and `update_recently_deleted` conflict types.
`insert_or_error`	Tries to build new row from available information sent by origin and insert it. If there isn't enough information available to build full row, throws an error and stops the replication. If there isn't enough information available to build full row, throws an error and stops the replication. Can be used for `update_missing` and `update_recently_deleted` conflict types.
`ignore`	Ignores any missing target column and continues processing.Can be used for the `target_column_missing` conflict type.
`ignore_if_null`	Ignores a missing target column if the extra column in the remote row contains a NULL value. Otherwise, throws an error and stops replication. Can be used for the `target_column_missing` conflict type.
`use_default_value`	Fills the missing column value with the default (including NULL if that's the column default) and continues processing. Any error while processing the default or violation of constraints (that is, 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 shows the conflict types each conflict resolver can handle.

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

Default conflict resolvers

Conflict type	Resolver
insert_exists	update_if_newer
update_differing	update_if_newer
update_origin_change	update_if_newer
update_missing	insert_or_skip
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 (see note)	skip_if_recently_dropped
apply_error_ddl	error

> **target_table_missing**
>
> This conflict type isn't detected on community Postgresql. If the target table is missing, it causes an error and halts replication. EDB Postgres servers detect and handle missing target tables and can invoke the resolver.

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:

Resolution	Description
`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 diagnosing and handling multi-master conflicts, PGD, by default, logs every conflict into the `bdr.conflict_history` table. You can change this behavior with more granularity using bdr.alter_node_set_log_config.

33.9 Conflict functions

`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 with the `ROLLBACK` of the transaction, and the changes are visible to the current transaction.

Only the owner of the `relation` can execute the `bdr.alter_table_conflict_detection` function unless `bdr.backwards_compatibility` is set to 30618 or less.

> **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 and gives a warning, as long as warnings aren't disabled with `bdr.camo_enable_client_warnings`.

`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.

`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

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.

This function is transactional. You can roll back the changes, and they're 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

If `log_to_table` is set to true, conflicts are logged to a table. 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. Autopartition creates between 7 and 14 partitions in advance. bdr_superuser can change these defaults.

Since conflicts generated for all tables managed by PGD are logged to this table, it's important to ensure that only legitimate users can read the conflicted data. PGD 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 PGD also enforces a stricter policy on the conflict log table.

The predefined 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`. This view doesn't contain user data, allowing monitoring of any conflicts logged.

33.10 Replication set management

> **Replication management and DDL**
>
> 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 PGD 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 PGD 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 PGD group default replication set to replicate DDL or the PGD 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 subscriptions 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 must be a 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 behavior allows for calling the function not only on the node that's part of the PGD 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 PGD.) This is currently true even if the DDL filters configuration otherwise prevents 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.

33.11 Replication set membership

`bdr.replication_set_add_table`

This function adds a table to a replication set.

This function adds a table to a replication set and starts replicating changes from the committing of the transaction that contains the call to the function. 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 PGD group default replication set is used.
- `columns` — Reserved for future use (currently does nothing and must be NULL).
- `row_filter` — SQL expression to use for filtering the replicated rows. If this expression isn't defined (that is, it's 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.

If the `row_filter` isn't NULL, the function takes a `DML` global lock on the relation that's being added to the replication set. 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 PGD 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.

33.12 DDL replication filtering

See also DDL replication filtering.

`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 function also affects replication of PGD admin functions.

This function 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 PGD 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 PGD functions to be filtered using `bdr.*`.

The `role_name` is used for matching against the current role that's 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 PGD group default replication set is used.
- `ddl_filter_name` — Name of the DDL filter. This name must be unique across the whole PGD 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 aren't taken into consideration (that is, only the exclusive filter is applied). When multiple exclusive filters match, an error is thrown. This parameter is useful for routing specific commands to a 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 PGD admin functions, define a filter like this:

```
SELECT bdr.replication_set_add_ddl_filter('mygroup', 'mygroup_admin',
$$bdr\..*$$);
```

To exclude everything except for 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 and one for indexes. This example 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 the DDL filtering settings.

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 PGD 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.

33.13 Testing and tuning commands

EDB Postgres Distributed has tools that help with testing and tuning your PGD clusters. For background, see Testing and tuning.

`pgd_bench`

Synopsis

A benchmarking tool for EDB Postgres Distributed deployments.

```
pgd_bench [OPTION]... [DBNAME] [DBNAME2]
```

`DBNAME` can be a conninfo string of the format: `"host=10.1.1.2 user=postgres dbname=master"`

See pgd_bench in Testing and tuning for examples of pgd_bench options and usage.

Options

The pgd_bench command is implemented as a wrapper around the pgbench command. This means that it shares many of the same options and created tables named `pgbench` as it performs its testing.

Options that are specific to pgd_bench include the following.

Setting mode

`-m` or `--mode`

The mode can be set to `regular`, `camo`, or `failover`. The default is `regular`.

- `regular` — Only a single node is needed to run pgd_bench.
- `camo` — A second node must be specified to act as the CAMO partner. (CAMO must be set up.)
- `failover` — A second node must be specified to act as the failover.

When using `-m failover`, an additional option `--retry` is available. This option instructs pgd_bench to retry transactions when there's a failover. The `--retry` option is automatically enabled when `-m camo` is used.

Setting GUC variables

`-o` or `--set-option`

This option is followed by `NAME=VALUE` entries, which are applied using the Postgres `SET` command on each server that pgd_bench connects to, and only those servers.

The other options are identical to the Postgres pgbench command. For details, see the PostgreSQL pgbench documentation.

The complete list of options (pgd_bench and pgbench) follow.

Initialization options

- `-i, --initialize` — Invoke initialization mode.
- `-I, --init-steps=[dtgGvpf]+` (default `"dtgvp"`) — Run selected initialization steps.
  - `d` — Drop any existing pgbench tables.
  - `t` — Create the tables used by the standard pgbench scenario.
  - `g` — Generate data client-side and load it into the standard tables, replacing any data already present.
  - `G` — Generate data server-side and load it into the standard tables, replacing any data already present.
  - `v` — Invoke `VACUUM` on the standard tables.
  - `p` — Create primary key indexes on the standard tables.
  - `f` — Create foreign key constraints between the standard tables.

- `-F, --fillfactor=NUM` — Set fill factor.
- `-n, --no-vacuum` — Don't run `VACUUM` during initialization.
- `-q, --quiet` — Quiet logging (one message every 5 seconds).
- `-s, --scale=NUM` — Scaling factor.
- `--foreign-keys` — Create foreign key constraints between tables.
- `--index-tablespace=TABLESPACE` — Create indexes in the specified tablespace.
- `--partition-method=(range|hash)` — Partition `pgbench_accounts` with this method. The default is `range`.
- `--partitions=NUM` — Partition `pgbench_accounts` into `NUM` parts. The default is `0`.
- `--tablespace=TABLESPACE` — Create tables in the specified tablespace.
- `--unlogged-tables` — Create tables as unlogged tables. (Note: Unlogged tables aren't replicated.)

Options to select what to run

- `-b, --builtin=NAME[@W]` — Add built-in script NAME weighted at W. The default is 1. Use `-b list` to list available scripts.
- `-f, --file=FILENAME[@W]` — Add script `FILENAME` weighted at W. The default is 1.
- `-N, --skip-some-updates` — Updates of pgbench_tellers and pgbench_branches. Same as `-b simple-update`.
- `-S, --select-only` — Perform SELECT-only transactions. Same as `-b select-only`.

Benchmarking options

- `-c, --client=NUM` — Number of concurrent database clients. The default is 1.
- `-C, --connect` — Establish new connection for each transaction.
- `-D, --define=VARNAME=VALUE` — Define variable for use by custom script.
- `-j, --jobs=NUM` — Number of threads. The default is 1.
- `-l, --log` — Write transaction times to log file.
- `-L, --latency-limit=NUM` — Count transactions lasting more than NUM ms as late.
- `-m, --mode=regular|camo|failover` — Mode in which to run pgbench. The default is `regular`.
- `-M, --protocol=simple|extended|prepared` — Protocol for submitting queries. The default is `simple`.
- `-n, --no-vacuum` — Don't run `VACUUM` before tests.
- `-o, --set-option=NAME=VALUE` — Specify runtime `SET` option.
- `-P, --progress=NUM` — Show thread progress report every NUM seconds.
- `-r, --report-per-command` — Latencies, failures, and retries per command.
- `-R, --rate=NUM` — Target rate in transactions per second.
- `-s, --scale=NUM` — Report this scale factor in output.
- `-t, --transactions=NUM` — Number of transactions each client runs. The default is 10.
- `-T, --time=NUM` — Duration of benchmark test, in seconds.
- `-v, --vacuum-all` — Vacuum all four standard tables before tests.
- `--aggregate-interval=NUM` — Data over NUM seconds.
- `--failures-detailed` — Report the failures grouped by basic types.
- `--log-prefix=PREFIX` — Prefix for transaction time log file. The default is `pgbench_log`.
- `--max-tries=NUM` — Max number of tries to run transaction. The default is `1`.
- `--progress-timestamp` — Use Unix epoch timestamps for progress.
- `--random-seed=SEED` — Set random seed (`time`, `rand`, `integer`).
- `--retry` — Retry transactions on failover. Used with `-m`.
- `--sampling-rate=NUM` — Fraction of transactions to log, for example, 0.01 for 1%.
- `--show-script=NAME` — Show built-in script code, then exit.
- `--verbose-errors` — Print messages of all errors.

Common options:

- `-d, --debug` — Print debugging output.
- `-h, --host=HOSTNAME` — Database server host or socket directory.
- `-p, --port=PORT` — Database server port number.
- `-U, --username=USERNAME` — Connect as specified database user.
- `-V, --version` — Output version information, then exit.
- `-?, --help` — Show help, then exit.

33.14 Global sequence management interfaces

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

The following functions are considered to be `DDL`, so DDL replication and global locking applies to them.

Sequence functions

`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.

PGD 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 PGD 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 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, you 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, you 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. See also KSUUID in the sequences documentation.

`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). Deprecated - See KSUUID in the sequences documentation for details.

`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.

33.15 Autopartition

Autopartition allows you to split tables into several partitions. For more information, see Scaling.

`bdr.autopartition`

The `bdr.autopartition` function configures automatic RANGE 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` — The 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 create in a single go after 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 only two more partitions remain. Each partition can hold 1 billion orders.

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

`bdr.drop_autopartition`

Use `bdr.drop_autopartition()` to drop the autopartitioning 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.

`bdr.autopartition_wait_for_partitions`

Partition creation is an asynchronous process. AutoPartition provides a set of functions to wait for the partition to be created, locally or on all nodes.

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 waits only 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 PGD 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, upperbound
text);
```

Parameters

- `relation` — Name or Oid of a table.
- `upperbound` — Partition key column value.

`bdr.autopartition_wait_for_partitions_on_all_nodes`

Synopsis

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

Parameters

- `relation` — Name or Oid of a table.
- `upperbound` — Partition key column value.

`bdr.autopartition_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.

`bdr.autopartition_enable`

Use `bdr.autopartition_enable` to enable AutoPartitioning on the given table. If AutoPartitioning is already enabled, then no action occurs. See `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.

`bdr.autopartition_disable`

Use `bdr.autopartition_disable` to disable AutoPartitioning on the given table. If AutoPartitioning is already disabled, then no action occurs.

Synopsis

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

Parameters

- `relname` — Name of the relation to disable AutoPartitioning.

Internal functions

`bdr.autopartition_create_partition`

AutoPartition uses an internal function `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` — Lower bound of the partition.
- `upperb` — Upper bound of the partition.
- `nodes` — List of nodes that the new partition resides on. This parameter is internal to PGD and reserved for future use.

Notes

This is an internal function used by AutoPartition for partition management. We recommend that you don't use the function directly.

`bdr.autopartition_drop_partition`

AutoPartition uses an internal function `bdr.autopartition_drop_partition` to drop a partition that's no longer required, as per the data-retention policy. 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 function places a DDL lock on the parent table before using `DROP TABLE` on the chosen partition table. This function is an internal function used by AutoPartition for partition management. We recommend that you don't use the function directly.

33.16 Stream triggers reference

> **SeeAlso**
>
> Stream Triggers for an introduction to Stream Triggers.

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 PGD.

In PL/pgSQL, you can use the predefined variables and functions that follow:

- Row variables
- Row Information functions
  - `bdr.trigger_get_row`
  - `bdr.trigger_get_committs`
  - `bdr.trigger_get_xid`
  - `bdr.trigger_get_type`
  - `bdr.trigger_get_conflict_type`
  - `bdr.trigger_get_origin_node_id`
  - `bdr.ri_fkey_on_del_trigger`

Creating and dropping stream triggers is managed through the manipulation interfaces:

- Manipulation interfaces
  - `bdr.create_conflict_trigger`
  - `bdr.create_transform_trigger`
  - `bdr.drop_trigger`

33.16.1 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 PGD 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 PGD 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.

33.16.2 Stream triggers row 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 the descriptions of the 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 the descriptions of the 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 the descriptions of the 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 the descriptions of the 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()
```

33.16.3 Stream triggers row variables

`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.

33.17 Internal catalogs and views

Catalogs and views are listed here in alphabetical order.

`bdr.autopartition_partitions`

An internal catalog table that stores information about the partitions created by the autopartitioning feature.

`bdr.autopartition_partitions` columns

Name	Type	Description
ap_parent_relid	oid	OID for relation
ap_part_relname	name	Name of created relation
ap_part_created_at	timestamp with time zone	Creation timestamp
ap_part_migrated_at	timestamp with time zone	Migration timestamp
ap_part_dropped_at	timestamp with time zone	Timestamp when dropped

`bdr.autopartition_rules`

An internal catalog table that stores information about the autopartitioning rules.

`bdr.autopartition_rules` columns

Name	Type	Description
ap_partition_relid	oid	
ap_partition_relname	name	
ap_partition_schemaname	name	
ap_partition_increment_kind	"char"	
ap_secondary_tablespace	oid	
ap_maximum_advance_partitions	integer	
ap_is_autoscaled	boolean	
ap_latest_partitions	integer	
ap_enabled	boolean	
ap_migrate_after_period	interval	
ap_data_retention_period	interval	
ap_last_triggered	timestamp with time zone	
ap_partition_increment_value	text	
ap_partition_autocreate_expr	text	
ap_partition_initial_lowerbound	text	
ap_partition_last_upperbound	text	
ap_is_local	boolean	
ap_partition_min_upperbound	text	

`bdr.ddl_epoch`

An internal catalog table holding state per DDL epoch.

`bdr.ddl_epoch` columns

Name	Type	Description
ddl_epoch	int8	Monotonically increasing epoch number
origin_node_id	oid	Internal node ID of the node that requested creation of this epoch
epoch_consume_timeout	timestamptz	Timeout of this epoch
epoch_consumed	boolean	Switches to true as soon as the local node has fully processed the epoch
epoch_consumed_lsn	boolean	LSN at which the local node has processed the epoch

`bdr.event_history`

Internal catalog table that tracks cluster membership events for a given PGD node. Specifically, it tracks:

- Node joins (to the cluster)
- Raft state changes (that is, whenever the node changes its role in the consensus protocol - leader, follower, or candidate to leader); seeMonitoring Raft consensus
- Whenever a worker has errored out (see bdr.workers and Monitoring PGD replication workers)

`bdr.event_history` columns

Name	Type	Description
event_node_id	oid	ID of the node to which the event refers
event_type	int	Type of the event (a node, raft, or worker-related event)
event_sub_type	int	Subtype of the event, that is, if it's a join, a state change, or an error
event_source	text	Name of the worker process where the event was sourced
event_time	timestamptz	Timestamp at which the event occurred
event_text	text	Textual representation of the event (for example, the error of the worker)
event_detail	text	A more detailed description of the event (for now, only relevant for worker errors)

`bdr.event_summary`

A view of the `bdr.event_history` catalog that displays the information in a more human-friendly format. Specifically, it displays the event types and subtypes as textual representations rather than integers.

`bdr.local_leader_change`

This is a local cache of the recent portion of leader change history. It has the same fields as `bdr.leader`, except that it is an ordered set of (node_group_id, leader_kind, generation) instead of a map tracking merely the current version.

`bdr.node_config`

An internal catalog table with per-node configuration options.

`bdr.node_config` columns

Name	Type	Description
node_id	oid	Node ID
node_route_priority	int	Priority assigned to this node
node_route_fence	boolean	Switch to fence this node
node_route_writes	boolean	Switch to allow writes
node_route_reads	boolean	Switch to allow reads
node_route_dsn	text	Interface of this node

`bdr.node_config_summary`

A view of the `bdr.node_config` catalog that displays the information in a more human-readable format.

`bdr.node_config_summary` columns

Name	Type	Description
node_name	text	The name of this node
node_id	oid	Node ID
node_route_priority	int	Priority assigned to this node
node_route_fence	boolean	Switch to fence this node
node_route_writes	boolean	Switch to allow writes
node_route_reads	boolean	Switch to allow reads
node_route_dsn	text	Interface of this node
effective_route_dsn	text	Full DSN of this node

`bdr.node_group_config`

An internal catalog table with per-node group configuration options.

`bdr.node_group_config` columns

Name	Type	Description
node_group_id	oid	Node group ID
route_writer_max_lag	bigint	Maximum write lag accepted
route_reader_max_lag	bigint	Maximum read lag accepted
route_writer_wait_flush	boolean	Switch if we need to wait for the flush

`bdr.node_group_routing_config_summary`

Per-node-group routing configuration options.

`bdr.node_group_routing_config_summary` columns

Name	Type	Description
node_group_name	name	Node group name
location	name	Node group location
enable_proxy_routing	boolean	Group proxy routing enabled?
node_group_type	text	Node group type (one of "global", "data", or "subscriber-only")
route_writer_max_lag	bigint	Maximum write lag accepted
route_reader_max_lag	bigint	Maximum read lag accepted
route_writer_wait_flush	boolean	Wait for flush

`bdr.node_group_routing_info`

An internal catalog table holding current routing information for a proxy.

`bdr.node_group_routing_info` columns

Name	Type	Description
node_group_id	oid	Node group ID.
write_node_id	oid	Current write node.
prev_write_node_id	oid	Previous write node.
read_node_ids	oid[]	List of read-only nodes IDs.
record_version	bigint	Record version. Incremented by 1 on every material change to the routing record.
record_ts	timestamptz	Timestamp of last update to record_version.
write_leader_version	bigint	Write leader version. Copied from record_version every time write_node_id is changed.
write_leader_ts	timestamptz	Write leader timestamp. Copied from record_ts every time write_node_id is changed.
read_nodes_version	bigint	Read nodes version. Copied from record_version every time read_node_ids list is changed.
read_nodes_ts	timestamptz	Read nodes timestamp. Copied from record_tw every time read_node_ids list is changed.

`bdr.node_group_routing_summary`

A view of `bdr.node_group_routing_info` catalog that shows the information in more friendly way.

`bdr.node_group_routing_summary` columns

Name	Type	Description
node_group_name	name	Node group name
write_lead	name	Current write lead

Name	Type	Description
previous_write_lead	name	Previous write lead
read_nodes	name[]	Current read-only nodes

`bdr.node_routing_config_summary`

A friendly view of the per-node routing configuration options. Shows the node name rather than the oid and shorter field names.

`bdr.node_routing_config_summary` columns

Name	Type	Description
node_name	name	Node name
route_priority	int	Priority assigned to this node
route_fence	boolean	Switch to fence this node
route_writes	boolean	Switch to allow writes
route_reads	boolean	Switch to allow reads
route_dsn	text	Interface of this node

`bdr.proxy_config`

An internal catalog table holding proxy specific configurations.

`bdr.proxy_config` columns

Name	Type	Description
proxy_name	name	Name of the proxy
node_group_id	oid	Node group ID that this proxy uses
listen_port	int	Port that the proxy uses for read-write connections (setting to 0 disables port)
max_client_conn	int	Number of maximum read-write client connections that the proxy accepts
max_server_conn	int	Number of maximum read-write connections that the server accepts
server_conn_timeout	interval	Timeout for the read-write server connections
server_conn_keepalive	interval	Interval between the server connection keep-alive
fallback_group_timeout	interval	Timeout needed for the fallback
fallback_group_ids	oid[]	List of group IDs to use for the fallback
listen_addrs	text[]	Listen address
read_listen_port	int	Port that the proxy uses for read-only connections (setting to 0 disables port)
read_max_client_conn	int	Number of maximum read-only client connections that the proxy accepts
read_max_server_conn	int	Number of maximum read-only connections that the server accepts
read_server_conn_timeout	interval	Timeout for the server read-only connections
read_server_conn_keepalive	interval	Interval between the server read-only connection keep-alive
read_listen_addrs	text[]	Listen address for read-only connections
read_consensus_grace_period	interval	Duration for which proxy continues to route even upon loss of consensus

`bdr.proxy_config_summary`

A friendly view of per-proxy, instance-specific configuration options.

`bdr.proxy_config_summary` columns

Name	Type	Description
proxy_name	name	Name of the proxy
node_group_name	name	Node group name that this proxy uses
listen_port	int	Port that the proxy uses for read-write connections (setting to -1 disables port)
max_client_conn	int	Number of maximum read-write client connections that the proxy accepts
max_server_conn	int	Number of maximum read-write connections that the server accepts
server_conn_timeout	interval	Timeout for the read-write server connections
server_conn_keepalive	interval	Interval between the server connection keep-alive
node_group_enable_proxy_routing	boolean	Does the group the proxy is in enable proxy routing?
node_group_location	name	The group's location value
fallback_group_timeout	interval	Timeout needed for the fallback
fallback_group_ids	oid[]	List of group IDs to use for the fallback
listen_addrs	text[]	Listen address
read_listen_port	int	Port that the proxy uses for read-only connections (setting to -1 disables port)
read_max_client_conn	int	Number of maximum read-only client connections that the proxy accepts
read_max_server_conn	int	Number of maximum read-only connections that the server accepts
read_server_conn_timeout	interval	Timeout for the server read-only connections
read_server_conn_keepalive	interval	Interval between the server read-only connection keep-alive
read_listen_addrs	text[]	Listen address for read-only connections
read_consensus_grace_period	interval	Duration for which proxy continues to route even upon loss of consensus

`bdr.sequence_kind`

An internal state table storing the type of each non-local 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)

33.18 Internal system functions

The following are internal system functions. Many are used when creating various views. We recommend that you do not use the functions directly but instead use the views that they serve.

General internal functions

`bdr.bdr_get_commit_decisions`

Convenience routine to inspect shared memory state.

Synopsis

```
bdr.bdr_get_commit_decisions(dbid OID,
            origin_node_id
OID,
            origin_xid xid,
            local_xid xid,
            decision
"char",
            decision_ts
timestamptz,
            is_camo boolean)
```

`bdr.bdr_track_commit_decision`

Save the transaction commit status in the shared memory hash table. This dunction is used by the upgrade scripts to transfer commit decisions saved in bdr.node_pre_commit catalog to the shared memory hash table. The transaction commit status will also be logged to the WAL and hence can be reloaded from WAL.

Synopsis

```
bdr.bdr_track_commit_decision(OID, xid, xid, "char", timestamptz, boolean);
```

`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

Parameter	Description

Parameter	Description
`key`	An arbitrary key to fetch.

Notes

This function is an internal function, mainly used by HARP.

> **Warning**
>
> Don't use this function in user applications.

`bdr.consensus_kv_store`

Stores value in the consistent KV Store.

Returns the timestamp of the value expiration time. This function depends on `ttl`. If `ttl` is `NULL`, then this function 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

Parameter	Description
`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.decode_message_payload`

PGD message payload function that decodes the payloads of consensus messages to a more human-readable output. Used primarily by the `bdr.global_consensus_journal_details` debug view.

`bdr.decode_message_response_payload`

PGD message payload function that decodes the payloads of responses to consensus messages to a more human-readable output. Used primarily by the `bdr.global_consensus_journal_details` debug view.

`bdr.difference_fix_origin_create`

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

`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_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 has 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 a future release.

Synopsis

```
bdr.difference_fix_session_setup()
```

`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 function requires bdr_superuser rather than postgres superuser permissions.

Synopsis

```
bdr.difference_fix_xact_set_avoid_conflict()
```

`bdr.drop_node`

Drops a node's metadata.

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

- The local node, in which case it removes all the node metadata, including information about remote nodes.
- A remote node, in which case it removes only metadata for that specific node.

> **When to use `bdr.drop_node()`**
>
> Do not use `bdr.drop_node()` to drop node metadata and reuse node names. PGD can reuse existing node names providing the node name belongs to a node in a `PARTED` state. Use `bdr.part_node()` to remove the original node and place it in a `PARTED` state.
>
> Use of this internal function is limited to:
>
> - When you're instructed to by EDB Technical Support.
> - Where you're specifically instructed to in the documentation.
>
> Use `bdr.part_node` to remove a node from a PGD group. That function sets the node to `PARTED` state and enables reuse of the node name.

Synopsis

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

Parameters

Parameter	Description
`node_name`	Name of an existing node.
`cascade`	Deprecated, will be removed in a future release.
`force`	Circumvents all sanity checks and forces the removal of all metadata for the given PGD 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 function, 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 is 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**
>
> PGD 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.get_global_locks`

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_node_conflict_resolvers`

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

`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_alter_sequence_set_kind`

A function previously used internally for replication of the various function calls. No longer used by the current version of PGD. Exists only for backward compatibility during rolling upgrades.

`bdr.internal_replication_set_add_table`

A function previously used internally for replication of the various function calls. No longer used by the current version of PGD. Exists only for backward compatibility during rolling upgrades.

`bdr.internal_replication_set_remove_table`

A function previously used internally for replication of the various function calls. No longer used by the current version of PGD. 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 PGD 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`

Displays information for the local node needed by the PGD group reconfiguration internal mechanisms.

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

`bdr.msgb_connect`

Connects to the connection pooler of another node. Used by the consensus protocol.

`bdr.msgb_deliver_message`

Sends messages to another node's connection pooler. Used by the consensus protocol.

`bdr.node_catchup_state_name`

Converts catchup state code in name.

Synopsis

```
bdr.node_catchup_state_name(catchup_state oid);
```

Parameters

Parameter	Description
`catchup_state`	Oid code of the catchup state.

`bdr.node_kind_name`

Returns human-friendly name of the node kind (data|standby|witness|subscriber-only).

`bdr.peer_state_name`

Transforms the node state (`node_state`) into a textual representation. Used mainly to implement the `bdr.node_summary` view.

`bdr.pg_xact_origin`

Returns the origin id of a given transaction.

Synopsis

```
bdr.pg_xact_origin(xmin xid)
```

Parameters

Parameter	Description
`xid`	Transaction id whose origin is returned.

`bdr.request_replay_progress_update`

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

`bdr.reset_relation_stats`

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

`bdr.reset_subscription_stats`

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

`bdr.resynchronize_table_from_node`

Resynchronizes the relation from a remote node.

Synopsis

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

Parameters

Parameter	Description
`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 re-creating foreign key constraints

After running the function on a referenced table, if the referenced column data no longer matches the referencing column values, the function 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.seq_currval`

Part of the internal implementation of global sequence manipulation.

Invoked automatically when `currval()` is called on a galloc or snowflakeid sequence.

`bdr.seq_lastval`

Part of the internal implementation of global sequence manipulation.

Invoked automatically when `lastval()` is called on a galloc or snowflakeid sequence.

`bdr.seq_nextval`

Part of the internal implementation of global sequence increments.

Invoked automatically when `nextval()` is called on a galloc or snowflakeid sequence

`bdr.show_subscription_status`

Retrieves information about the subscription status. Used mainly to implement the `bdr.subscription_summary` view.

`bdr.show_workers`

Information related to the bdr workers.

Synopsis

```
bdr.show_workers(
   worker_pid int,
```

```
    worker_role
int,
    worker_role_name
text,
    worker_subid oid)
```

`bdr.show_writers`

Function used in the `bdr.writers` view.

Task manager functions

`bdr.taskmgr_set_leader`

Requests the given `node` to be the task manager leader node. The leader node is responsible for creating new tasks. (Currently only autopartition makes use of this facility.) A witness node, a logical standby, or a subscriber-only node can't become a leader. Such requests will fail with an error.

Synopsis

```
bdr.taskmgr_set_leader(node name, wait_for_completion boolean DEFAULT
true);
```

`bdr.taskmgr_get_last_completed_workitem`

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

Synopsis

```
bdr.taskmgr_get_last_completed_workitem();
```

`bdr.taskmgr_work_queue_check_status`

Lets you see the status of the background workers that are doing their job to generate and finish the tasks.

The status can be seen through these views:

- `bdr.taskmgr_work_queue_local_status`
- `bdr.taskmgr_work_queue_global_status`

Synopsis

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

Parameters

Parameter	Description
`workid`	The key of the task.
`local`	Check the local status only.

Notes

Taskmgr workers are always running in the background, even before the `bdr.autopartition` function is called for the first time. If an invalid `workid` is used, the function returns `unknown`. `In-progress` is the typical status.

`bdr.pglogical_proto_version_ranges`

Internal function for diagnostic use only.

`bdr.get_min_required_replication_slots`

Internal function intended for use by PGD-CLI.

`bdr.get_min_required_worker_processes`

Internal function intended for use by PGD-CLI.

`bdr.stat_get_activity`

Internal function underlying view `bdr.stat_activity`. Do not use directly. Use the `bdr.stat_activity` view instead.

`bdr.worker_role_id_name`

Internal helper function used when generating view `bdr.worker_tasks`. Do not use directly. Use the `bdr.worker_tasks` view instead.

`bdr.lag_history`

Internal function used when generating view `bdr.node_replication_rates`. Do not use directly. Use the `bdr.node_replication_rates` view instead.

`bdr.get_raft_instance_by_nodegroup`

Internal function used when generating view `bdr.group_raft_details`. Do not use directly. Use the `bdr.group_raft_details` view instead.

`bdr.monitor_camo_on_all_nodes`

Internal function used when generating view `bdr.group_camo_details`. Do not use directly. Use the `bdr.group_camo_details` view instead.

`bdr.monitor_raft_details_on_all_nodes`

Internal function used when generating view `bdr.group_raft_details`. Do not use directly. Use the `bdr.group_raft_details` view instead.

`bdr.monitor_replslots_details_on_all_nodes`

Internal function used when generating view `bdr.group_replslots_details`. Do not use directly. Use the `bdr.group_replslots_details` view instead.

`bdr.monitor_subscription_details_on_all_nodes`

Internal function used when generating view `bdr.group_subscription_summary`. Do not use directly. Use the `bdr.group_subscription_summary` view instead.

`bdr.monitor_version_details_on_all_nodes`

Internal function used when generating view `bdr.group_versions_details`. Do not use directly. Use the `bdr.group_versions_details` view instead.

`bdr.node_group_member_info`

Internal function used when generating view `bdr.group_raft_details`. Do not use directly. Use the `bdr.group_raft_details` view instead.

33.19 Column-level conflict functions

`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`.

34 PGD compatibility by PostgreSQL version

The following table shows the major versions of PostgreSQL and each version of EDB Postgres Distributed (PGD) they are compatible with.

Postgres Version	PGD 5	PGD 4	PGD 3.7	PGD 3.6
16	5.3+			
15	5			
14	5	4		
13	5	4	3.7	
12	5	4	3.7	
11			3.7	3.6
10				3.6