



EDB

EDB Postgres Distributed

Release 4.0.0

EDB

Jun 06, 2022

CONTENTS

1	Release notes	3
1.1	Release notes for EDB Postgres Distributed version 4	3
2	‘Known issues’	5
2.1	Data Consistency	5
2.2	List of Issues	5
2.3	List of Limitations	6
3	Key components	7
3.1	Postgres server	7
3.2	Postgres extensions	7
3.2.1	BDR	7
4	Terminology	9
4.1	Asynchronous replication	9
4.2	Availability	9
4.3	CAMO or commit-at-most-once	9
4.4	Clustering	9
4.5	Data sharding	10
4.6	Eager Replication for BDR	10
4.7	Eventual consistency	10
4.8	Failover	10
4.9	Horizontal scaling or scale out	10
4.10	Logical replication	10
4.11	Nines	10
4.12	Node	11
4.13	Physical replication	11
4.14	Read scalability	11
4.15	Recovery point objective (RPO)	11
4.16	Recovery time objective (RTO)	11
4.17	Single point of failure (SPOF)	11
4.18	Switchover	11
4.19	Synchronous replication	12
4.20	Two-phase commit (2PC)	12
4.21	Vertical scaling or scale up	12
4.22	Write scalability	12
5	Choosing your architecture	13
5.1	Standard EDB Always On architectures	13
5.2	Architecture details	14

5.3	Choosing your architecture	14
5.3.1	Always On Bronze (single active location - cloud region or on prem data center)	14
5.3.2	Always On Silver (single active location - cloud region or on prem data center)	15
5.3.3	Always On Gold (two active locations - cloud regions or on prem data centers)	17
5.3.4	Always On Platinum (two locations; fast HA restoration)	19
6	Choosing a Postgres distribution	21
7	Choosing durability	23
8	Other considerations	25
8.1	Deployment and sizing considerations	25
8.2	Clocks and timezones	25
9	Deployment options	27
9.1	TPAexec	27
9.1.1	Incremental changes	28
9.1.2	Extensible through Ansible	28
9.1.3	Cluster management	28
9.1.4	It's just Postgres	28
9.1.5	Getting started	28
9.1.5.1	Installing TPAexec	28
9.1.5.2	Using TPAexec	30
9.1.5.3	Deploying EDB Postgres Distributed	34
10	Upgrading	35
10.1	Upgrade Planning	35
10.1.1	Rolling Upgrade considerations	35
10.1.2	Rolling Server Software Upgrades	36
10.1.3	Rolling Upgrade Using Node Join	36
10.1.4	Upgrading a CAMO-Enabled Cluster	36
10.2	Upgrade Preparation	37
10.3	Server Software Upgrade	37
10.3.1	BDR Extension Upgrade	37
10.3.2	Stop Postgres	37
10.3.3	Upgrade Packages	37
10.3.4	Start Postgres	37
10.3.5	Postgres Upgrade	37
10.3.6	Minor Version Postgres Upgrade	37
10.3.7	Major Version Postgres Upgrade	38
10.4	Upgrade Check and Validation	38
10.5	Application Schema Upgrades	38
10.5.1	Rolling Application Schema Upgrades	38
10.5.1.1	In-place Postgres Major Version Upgrades	39
11	Compatibility matrix	43
12	Backup and recovery	45
12.1	Backup	45
12.1.1	pg_dump	45
12.1.2	Physical Backup	46
12.1.3	Eventual Consistency	46
12.1.4	Point-In-Time Recovery (PITR)	46
12.2	Restore	48
12.2.1	BDR Cluster Failure or Seeding a New Cluster from a Backup	48

12.2.2	Cleanup BDR Metadata	48
12.2.3	Cleanup of Replication Origins	48
12.2.4	Cleanup of Replication Slots	49
13	Monitoring	51
13.1	Monitoring Overview	51
13.2	Monitoring Node Join and Removal	52
13.3	Monitoring Replication Peers	53
13.3.1	Monitoring Outgoing Replication	53
13.3.2	Monitoring Incoming Replication	55
13.3.3	Monitoring WAL senders using LCR	55
13.4	Monitoring BDR Replication Workers	56
13.5	Monitoring BDR Writers	56
13.6	Monitoring Global Locks	57
13.7	Monitoring Conflicts	57
13.8	External Monitoring	58
13.8.1	bdr.set_node_location	58
13.8.2	Synopsis	58
13.8.3	Parameters	58
13.8.4	bdr.set_network_path_info	58
13.8.5	Synopsis	58
13.8.6	Parameters	59
13.8.7	bdr.network_monitoring view	59
13.9	Apply Statistics	59
13.10	Standard PostgreSQL Statistics Views	60
13.11	Monitoring BDR Versions	61
13.12	Monitoring Raft Consensus	61
13.13	Monitoring Replication Slots	63
13.14	Monitoring Transaction COMMITS	64
14	PGD CLI	65

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



Fig. 1: node diagram

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

Group Commit, or eager all node replication features.

RELEASE NOTES

The EDB Postgres Distributed documentation describes the Always On architectures supported in EDB Postgres Distributed 4. For details about individual product features see:- [BDR release notes](#) - [HARP release notes](#)

Version	Release Date
<i>Release notes for EDB Postgres Distributed version 4</i>	2022 May 17

1.1 Release notes for EDB Postgres Distributed version 4

EDB Postgres Distributed version 4 includes:

Feature	Description
Faster Postgres major version upgrade	Postgres Distributed is the only Postgres high availability technology that allows fully online major version upgrades. This functionality is improved to allow upgrades in-place, without migrating data between nodes. Even the largest clusters can now be upgraded to the latest major version within minutes.
Replication lag control	If you use asynchronous replication, you can configure a replication lag threshold after which the transactions get throttled. This allows you to keep a low RPO without incurring the latency impact on every transaction that comes with synchronous replication.
Distributed sequences by default	Instead of setting up custom sequence handling configuration on every node, you get distributed sequences configured automatically with the best kind of sequence for the data type used. The new SnowflakeID algorithm replaces Timeshard, which had constraints that prevented us from using it as a default.
Postgres Distributed Command Line Interface (CLI)	You can get information about your Postgres Distributed cluster right from the CLI, including the current state of the replication, consensus, and nodes. You can also query across the nodes of a cluster. Instead of relying on queries, you can use simplified commands in the CLI. See EDB Postgres Distributed Command Line Interface for more information.
Simplified synchronous replication configuration	Instead of adjusting multiple different Grand Unified Configurations (GUCs) for each node, you can use a SQL level interface to configure how transactions are replicated and what durability and visibility criteria to use, allowing for all nodes to be configured consistently from a single place. This allows (along with Group Commits) more functionality and flexibility around synchronous replication in the future.
Group Commit	You can have synchronous replication to require quorum within a Postgres Distributed group before committing a transaction, making EDB a more attractive alternative to Oracle DataGuard or Postgres Synchronous Replication. This allows more functionality around synchronous replication in the future, like Eager Majority.

‘KNOWN ISSUES’

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

2.1 Data Consistency

Please remember to read about *Monitoring Conflicts* to understand the implications of the asynchronous operation mode in terms of data consistency.

2.2 List of Issues

In this section we list a number of known issues that are tracked in BDR's ticketing system, that are expected to be re-solved in one of the future releases.

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

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

- If the resolver for the `update_origin_change` conflict is set to `skip`, and `synchronous_commit=remote_apply` is used, and concurrent updates of the same row are repeatedly applied on two different nodes, then one of the update statements might hang due to a deadlock with the BDR writer. As mentioned in the *Monitoring Conflicts* chapter, `skip` is not the default resolver for the `update_origin_change` conflict, and this combination is not intended to be used in production: it discards one of the two conflicting updates based on the order of arrival on that node, which is likely to cause a divergent cluster. In the rare situation that you do choose to use the `skip` conflict resolver, please note the issue with the use of the `remote_apply` mode.
- Decoding Worker feature does not work with CAMO/EAGER/Group Commit. Installations using CAMO/Eager/Group Commit must keep `enable_wal_decoder` disabled.
- Decoding Worker works only with the default replication sets
- Lag Control does not adjust commit delay in any way on a fully isolated node, i.e. in case all other nodes are unreachable or not operational. As soon as at least one node is connected, Replication Lag Control will pick up its work and adjust the BDR commit delay, again.

- For time based Lag Control, BDR currently uses the lag time (measured by commit timestamps) rather than the estimated catchup time that is based on historic apply rate.
- Changing the CAMO partners in a CAMO pair is not currently possible. It's only possible to add or remove a pair. Adding or removing a pair does not need a restart of Postgres or even a reload of the configuration.
- Group Commit cannot be combined with *CAMO* or *commit-at-most-once* or *Eager All Node replication*. Eager Replication currently only works by using the “global” BDR commit scope.
- Neither Eager replication nor Group Commit support `synchronous_replication_availability = 'async'`.
- Group Commit does not support support for a timeout of the commit after `bdr.global_commit_timeout`.
- Transactions using Eager Replication cannot yet execute DDL, nor do they support explicit two-phase commit. Note that the TRUNCATE command is allowed.
- Not all DDL can be run when either CAMO or Group Commit is used.
- Parallel apply is not currently supported in combination with Group Commit, please make sure to disable it when using Group Commit by either setting `num_writers` to 1 for the node group (using `bdr.alter_node_group_config`) or via the GUC `bdr.writers_per_subscription` (see Configuration of Generic Replication).
- There currently is no protection against altering or removing a commit scope. Running transactions in a commit scope that is concurrently being altered or removed may lead to the transaction blocking or replication stalling completely due to an error on the downstream node attempting to apply the transaction. Please ensure that any transactions using a specific commit scope have finished before altering or removing it.

2.3 List of Limitations

This section contains a (non-comprehensive) list of design limitations that are expected and are by design, and hence are not expected to be resolved in the future.

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

KEY COMPONENTS

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

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

3.1 Postgres server

Three different Postgres distributions can be used:

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

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

3.2 Postgres extensions

3.2.1 BDR

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

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

!!!note "pglogical3"

Older versions of BDR (3.6/3.7) depended on pglogical3 to provide the replication channel upon which BDR builds. It is no longer used.

HARP is connection management tool for a BDR cluster.

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

TERMINOLOGY

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

4.1 Asynchronous replication

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

4.2 Availability

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

4.3 CAMO or commit-at-most-once

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

4.4 Clustering

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

4.5 Data sharding

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

4.6 Eager Replication for BDR

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

4.7 Eventual consistency

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

4.8 Failover

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

4.9 Horizontal scaling or scale out

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

4.10 Logical replication

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

4.11 Nines

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

4.12 Node

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

4.13 Physical replication

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

4.14 Read scalability

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

4.15 Recovery point objective (RPO)

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

4.16 Recovery time objective (RTO)

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

4.17 Single point of failure (SPOF)

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

4.18 Switchover

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

4.19 Synchronous replication

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

4.20 Two-phase commit (2PC)

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

4.21 Vertical scaling or scale up

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

4.22 Write scalability

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

CHOOSING YOUR ARCHITECTURE

Always On architectures reflect EDB's recommended practices and help you to achieve the highest possible service availability in multiple configurations. These configurations range from single-location architectures to complex distributed systems that protect from hardware failures and data center failures. The architectures leverage EDB Postgres Distributed's multi-master capability and its ability to achieve 99.999% availability, even during maintenance operations.

You can use EDB Postgres Distributed for architectures beyond the examples described here. Use-case-specific variations have been successfully deployed in production. However, these variations must undergo rigorous architecture review first. Also, EDB's standard deployment tool for Always On architectures, TPAExec, must be enabled to support the variations before they can be supported in production environments.

5.1 Standard EDB Always On architectures

EDB has identified four standard Always On architectures:

- *Always On Bronze (single active location - cloud region or on prem data center)* : Single active location (data center or availability zone [AZ])
- *Always On Silver (single active location - cloud region or on prem data center)* : Single active location with redundant hardware to quickly restore failover capability and a backup in a disaster recovery (DR) location
- *Always On Gold (two active locations - cloud regions or on prem data centers)* : Two active locations
- *Always On Platinum (two locations; fast HA restoration)* : Two active locations with additional redundant hardware in a hot standby mode

All Always On architectures protect a progressively robust range of failure situations. For example, Always On Bronze protects against local hardware failure but doesn't provide protection from location (data center or AZ) failure. Always On Silver makes sure that a backup is kept at a different location, thus providing some protection in case of the catastrophic loss of a location. However, the database still must be restored from backup first, which might violate recovery time objective (RTO) requirements. Always On Gold provides two active locations connected in a multi-master mesh network, making sure that service remains available even in case a location goes offline. Finally, Always On Platinum adds redundant hot standby hardware in both locations to maintain local high availability in case of a hardware failure.

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

Increasing the availability guarantee drives additional cost for hardware and licenses, networking requirements, and operational complexity. Carefully consider your availability and compliance requirements before choosing an architecture.

5.2 Architecture details

EDB Postgres Distributed uses a *Monitoring Raft Consensus*-based consensus architecture. While regular database operations (insert, select, delete) don't require cluster-wide consensus, EDB Postgres Distributed benefits from an odd number of BDR nodes to make decisions that require consensus, such as generating new global sequences, or distributed DDL operations. Even the simpler architectures always have three BDR nodes, even if not all of them are storing data. Always On Gold and Platinum, which use two active locations, introduce a fifth BDR node as a witness node to support the RAFT requirements.

Applications connect to the standard Always On architectures by way of multi-host connection strings, where each HA-proxy server is a distinct entry in the multi-host connection string. Other connection mechanisms have been successfully deployed in production, but they're not part of the standard Always On architectures.

5.3 Choosing your architecture

All architectures provide the following: * Hardware failure protection * Zero downtime upgrades * Support for availability zones in public/private cloud

Use these criteria to help you to select the appropriate Always On architecture.

	Bronze	Silver	Gold	Platinum
Location failure protection	No (unless Barman is moved off-site)	Yes - Recovery from backup	Yes - instant failover to fully functional site	Yes - instant failover to fully functional site
Failover to DR or full DC	DR (if Barman is located offsite); NA otherwise	DR (if Barman is located offsite)	Full DC	Full DC
Fast local restoration of high availability after device failure	No; time to restore HA: (1) VM prov + (2) approx 60 min/500GB	Yes; three local data nodes allow to maintain HA after device failure	No; time to restore HA: (1) VM prov + (2) approx 60 min/500GB	Yes; logical standbys can quickly be promoted to master data nodes
Cross data center network traffic	Backup traffic only (if Barman is located offsite); none otherwise	Backup traffic only (if Barman is located offsite); none otherwise	Full replication traffic	Full replication traffic
License cost	2 data nodes	3 data nodes	4 data nodes	4 data nodes 2 logical standbys

5.3.1 Always On Bronze (single active location - cloud region or on prem data center)

The Always On Bronze architecture includes the following:

- Two BDR data nodes
- One BDR witness node that doesn't hold data but is used for consensus
- Two HARP-Proxy nodes for routing application traffic to the "lead" master
- One barman node for backup and recovery can be onsite or offsite

BDR and HARP-Proxy nodes should be spread across availability zones.

Always On Bronze

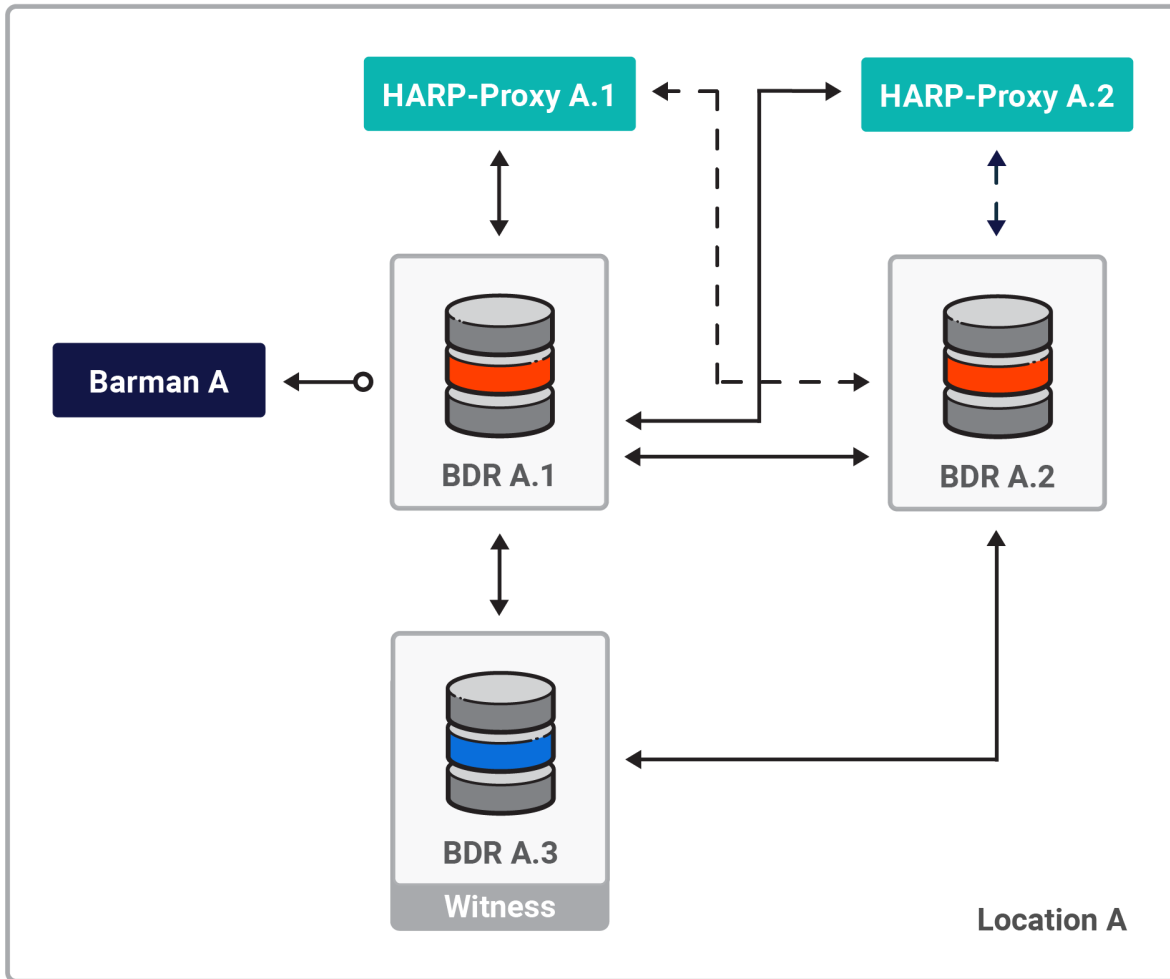


Fig. 1: Always On Bronze architecture

5.3.2 Always On Silver (single active location - cloud region or on prem data center)

The Always On Silver architecture includes the following:

- Three BDR data nodes
- Two HARP-Proxy nodes
- Barman offsite (offsite is optional but recommended)

BDR and HARP nodes should be spread across AZs. Barman can be located in location A.

Always On Silver

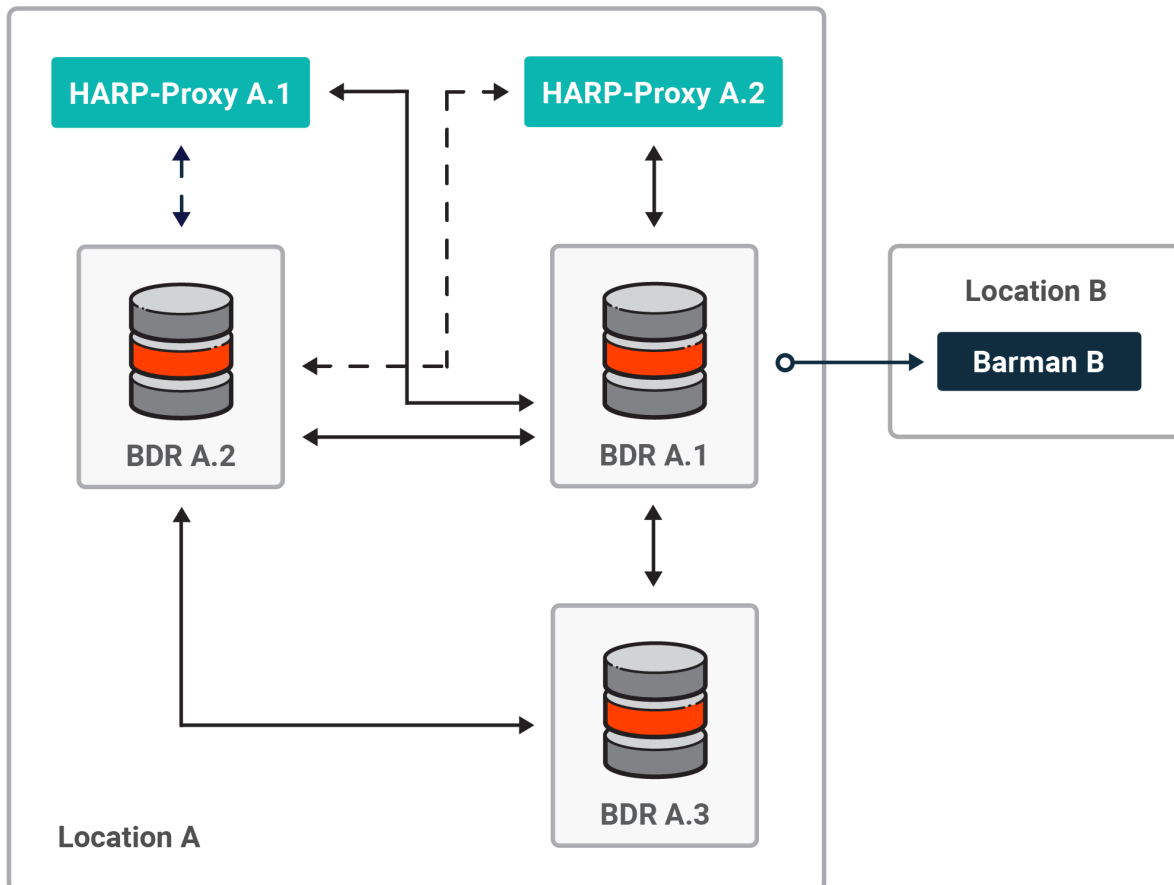


Fig. 2: Always On Silver architecture

5.3.3 Always On Gold (two active locations - cloud regions or on prem data centers)

This architecture favors local resiliency/redundancy first and remote locations only when and an entire location is offline or fails.

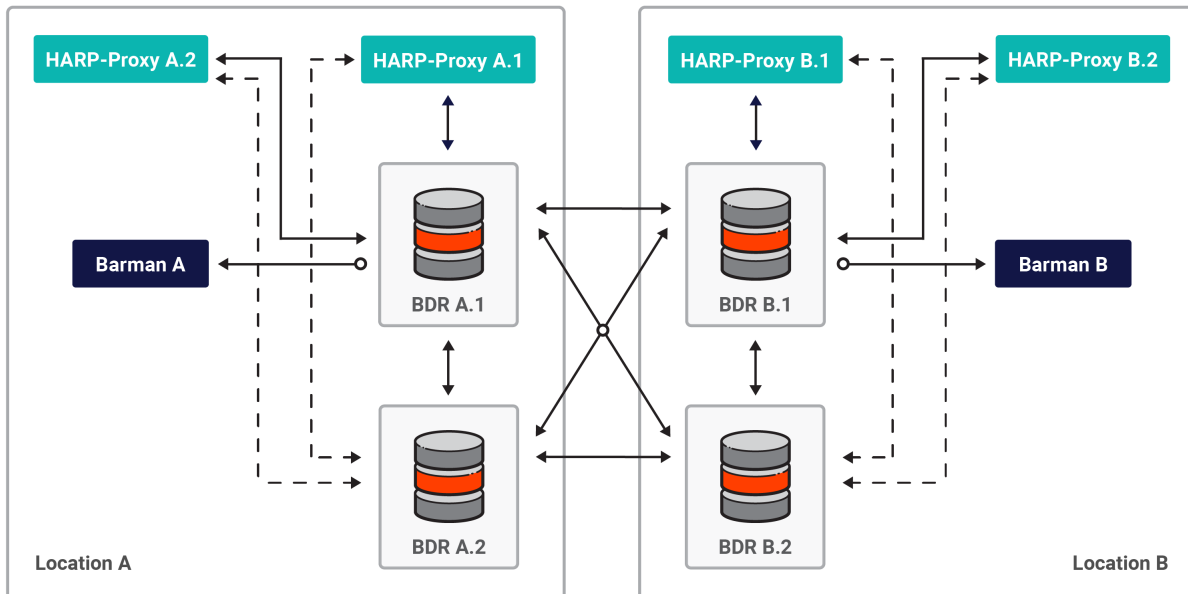
This architecture enables geo-distributed writes where no/low conflict handling is expected

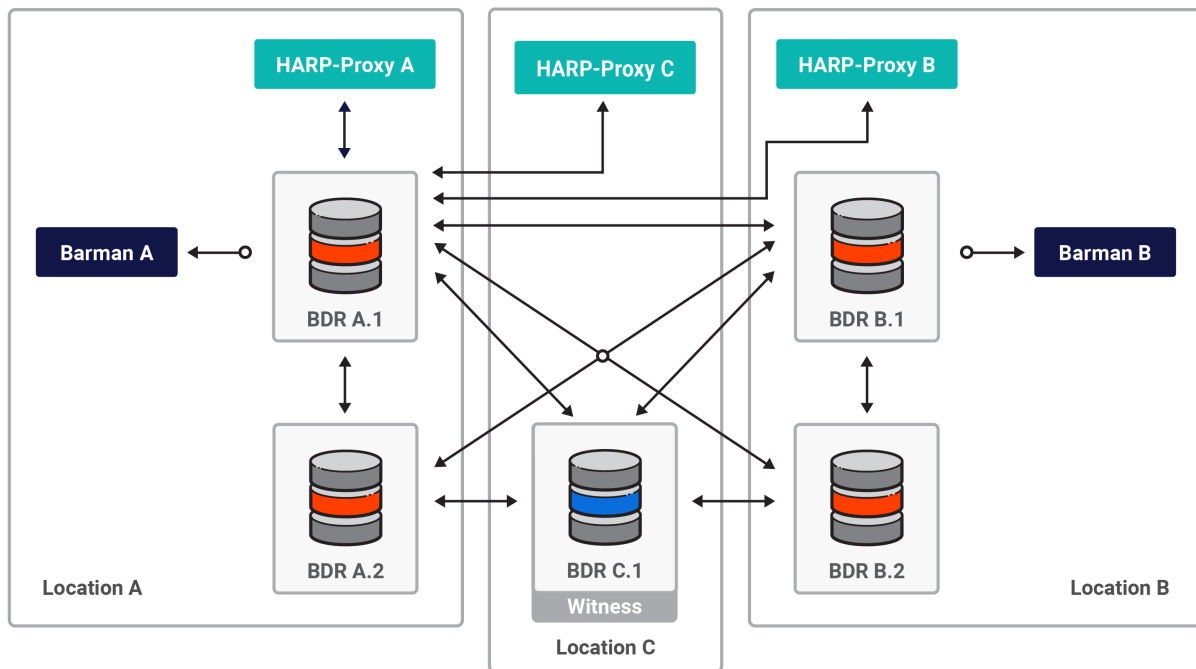
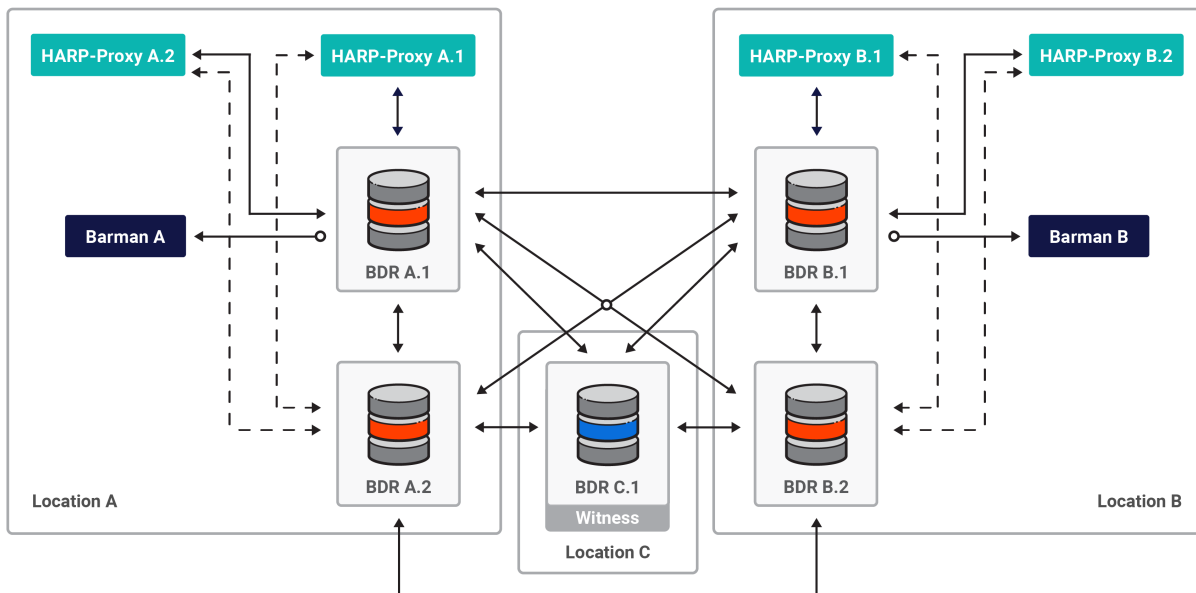
The Always On Gold architecture requires intervention to move between locations but all data will be replicated and available to the application when failover is initiated with full capacity.

The Always On Gold architecture includes the following:

- Four BDR data nodes (two in location A, two in location B)
- One BDR witness node in location C (optional but recommended)
- Four HARP-Proxy nodes (two in location A, two in location B)
- Two Barman nodes (one in location A, one in location B)

Always On Gold 2DC, 2 Lead Masters



Always On Gold 3DC, 1 Lead Master**Always On Gold 3DC, 2 Lead Masters**

5.3.4 Always On Platinum (two locations; fast HA restoration)

The Always On Platinum architecture includes the following:

- Four BDR data nodes (two in location A, two in location B)
- Two logical standby nodes (one in location A, one in location B)
- One witness node in location C (optional but recommended)
- Four HARP-Proxy nodes (two in location A, two in location B)
- Two Barman nodes (one in location A, one in location B)

Both locations have redundant hot standby hardware to maintain local high availability in case of a hardware failure without waiting to restore a BDR node from backup.

Always On Platinum

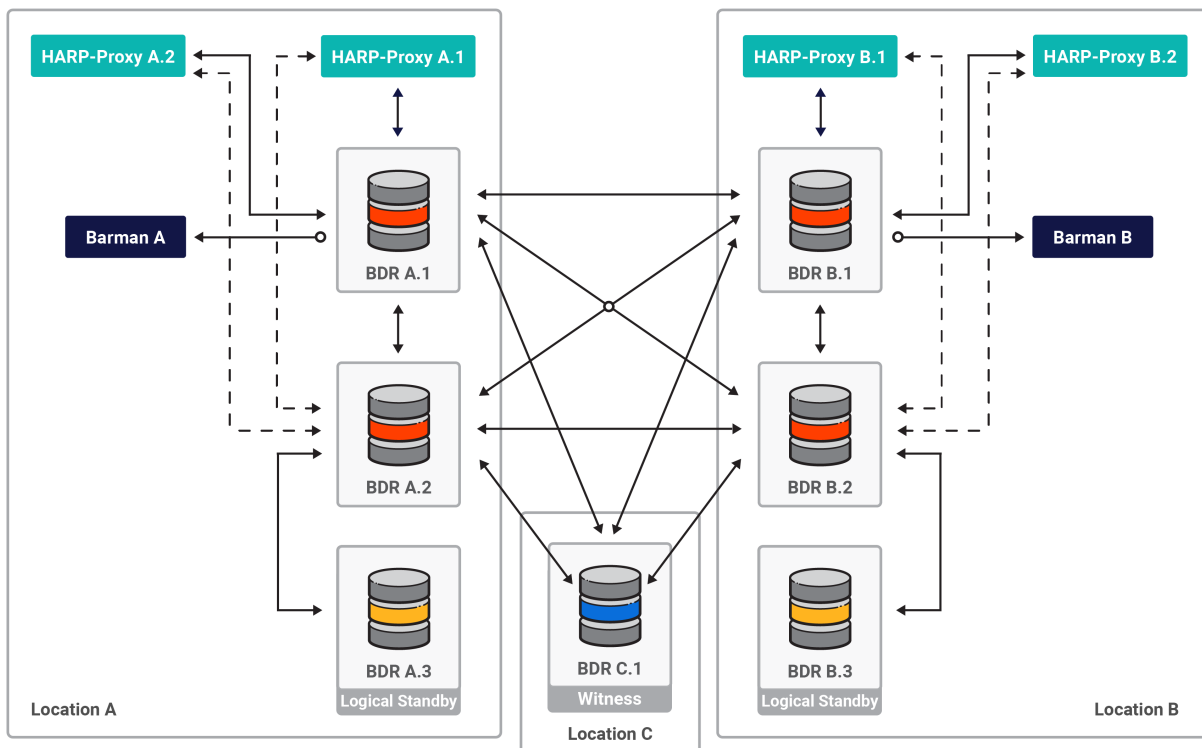


Fig. 3: Always On Platinum architecture

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 EDB Postgres Distributed features available depend on which Postgres distribution is used. Therefore, it is essential to adopt the Postgres distribution best suited to your business needs. For example, if having the feature “Commit At Most Once (CAMO)” is mission critical to a your use case, you should not adopt open source PostgreSQL because it does not have the core capabilities required to handle CAMO.

The following table lists features of EDB Postgres Distributed that are dependent on the Postgres distribution and version.

Feature	PostgreSQL	EDB Postgres Extended	EDB Postgres Advanced
Commit At Most Once (CAMO)	N	Y	14+
Eager Replication	N	Y	14+
Decoding Worker	N	13+	14+
Assessment Tooling	N	Y	14+
Lag Tracker	N	Y	14+
Lag Control	N	Y	14+
Timestamp Snapshots	N	Y	14+
Transaction Streaming	14+	13+	14+
Missing Partition Conflict	N	Y	14+
No need for UPDATE Trigger on tables with TOAST	N	Y	14+
Automatically hold back FREEZE	N	Y	14+

CHOOSING DURABILITY

EDB Postgres Distributed allows you to choose from several replication configurations based on your consistency, availability, and performance needs:

- Asynchronous
- Synchronous (using `synchronous_standby_names`)
- *Enable Commit At Most Once*
- *Eager Replication for BDR*
- Group Commit

For more information, see *Choosing durability* .

OTHER CONSIDERATIONS

Review these other considerations when planning your deployment.

8.1 Deployment and sizing considerations

For production deployments, EDB recommends a minimum of 12 cores for each Postgres data node and each logical standby. Witness nodes don't participate in the data replication operation and don't have to meet this requirement. Always size logical standbys exactly like the data nodes to avoid performance degradations in case of a node promotion. In production deployments, HARP proxy nodes require a minimum of four cores each. EDB recommends detailed benchmarking of performance requirements, working with EDB's Professional Services team.

For development purposes, don't assign Postgres data nodes fewer than two cores. The sizing of Barman nodes depends on the database size and the data change rate.

You can deploy Postgres data nodes, logical standbys, Barman nodes, and HARP proxy nodes on virtual machines or in a bare metal deployment mode. However, maintain anti-affinity between data nodes and HARP proxy nodes. Don't deploy multiple data nodes on VMs that are on the same physical hardware, as that reduces resiliency. Also don't deploy multiple HARP proxy nodes on VMs on the same physical hardware, as that, too, reduces resiliency.

You can deploy single HARP proxy nodes with single data nodes on the same physical hardware but should be deployed as VMs to ensure proper CPU and memory resource assignment.

8.2 Clocks and timezones

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

Server clocks should be synchronized using NTP or other solutions.

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

DEPLOYMENT OPTIONS

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

- TPAexec is an orchestration tool that uses Ansible to build Postgres clusters as specified by TPA (Trusted Postgres Architecture), a set of reference architectures that document how to set up and operate Postgres in various scenarios. TPA represents the best practices followed by EDB, and its recommendations are as applicable to quick testbed setups as to production environments.

Coming soon:

- BigAnimal is a fully managed database-as-a-service with built-in Oracle compatibility, running in your cloud account and operated by the Postgres experts. BigAnimal makes it easy to set up, manage, and scale your databases. The addition of extreme high availability support through EDB Postgres Distributed allows single-region Always On Gold clusters: two BDR groups in different availability zones in a single cloud region, with a witness node in a third availability zone.
- EDB Postgres Distributed for Kubernetes will be a Kubernetes operator is designed, developed, and supported by EDB that covers the full lifecycle of a highly available Postgres database clusters with a multi-master architecture, using BDR replication. It is based on the open source CloudNativePG operator, and provides additional value such as compatibility with Oracle using EDB Postgres Advanced Server and additional supported platforms such as IBM Power and OpenShift.

9.1 TPAexec

The standard way of deploying EDB Distributed Postgres in a self managed setting, including physical and virtual machines, both self-hosted and in the cloud (EC2), is to use EDB's deployment tool called TPAexec.

TPAexec is an orchestration tool that uses Ansible to build Postgres clusters as specified by Trusted Postgres Architecture (TPA), a set of reference architectures that documents how to set up and operate Postgres in various scenarios. TPA represents the best practices followed by EDB and its recommendations are as applicable to quick testbed setups as to production environments.

TPAexec packages are only available to EDB customers with an active BDR subscription. Please contact your account manager to request access.

9.1.1 Incremental changes

TPAexec is carefully designed so that provisioning, deployment, and testing are idempotent. You can run through them, make a change to `config.yml`, and run through the process again to deploy the change. If nothing has changed in the configuration or on the instances, then rerunning the entire process does not change anything either.

9.1.2 Extensible through Ansible

TPAexec supports a variety of configuration options, so you can do a lot just by editing `config.yml` and re-running `provision/deploy/test`. Should you need to go beyond what is already implemented, you can write hook scripts to extend the deployment process.

This mechanism places the full range of Ansible functionality at your disposal during every stage of the deployment. For example, any tasks in `hooks/pre-deploy.yml` are executed before the main deployment; and there are also `post-deploy` and many other hooks.

9.1.3 Cluster management

Once your cluster is up and running, TPAexec provides convenient cluster management functions, including configuration changes, switchover, and zero-downtime minor-version upgrades. These features make it easier and safer to manage your cluster than making the changes by hand.

9.1.4 It's just Postgres

TPAexec can create complex clusters with many features configured, but the result is just Postgres. The installation follows some conventions designed to make life simpler, but there is no hidden magic or anything standing in the way between you and the database. You can do everything on a TPA cluster that you could do on any other Postgres installation.

9.1.5 Getting started

See these topics to install TPAexec and use it to deploy your first EDB Postgres Distributed cluster:

- *[Installing TPAexec](#)*
- *[Using TPAexec](#)*
- *[Example: Deploying EDB Postgres Distributed](#)*

For more information, see *[TPAexec](#)*.

9.1.5.1 Installing TPAexec

The TPAexec install package is available from the EDB customer portal. You must have a subscription to access the repository and the TPAexec install package.

9.1.5.2 Using TPAexec

With TPAexec you configure, provision, and deploy your EDB Postgres Distributed clusters.

Configure

The `tpaexec configure` command generates a simple YAML configuration file to describe a cluster, based on the options you select. The configuration is ready for immediate use and you can modify it to better suit your needs. Editing the configuration file is the usual way to make any configuration changes to your cluster both before and after it's created.

The syntax is:

```
tpaexec configure <cluster_dir> --architecture <architecture_name> --layout <layout> ↵  
↵[options]
```

The required configuration options include:

Flags	Description
<code>-architecture</code>	Required. Set to <i>BDR-Always-ON</i> for EDB Postgres Distributed deployments.
<code>-layout</code>	Required. Specify one of the four supported architectures: bronze, silver, gold, and platinum. See Choosing your architecture for more information.
<code>-harp-consensus-protocol</code>	Required. Use <i>bdr</i> if your layout is bronze or silver. Use <i>etcd</i> if your layout is gold or platinum.

For example:

```
[tpa]$ tpaexec configure ~/clusters/speedy \  
      --architecture BDR-Always-ON \  
      --layout gold \  
      --harp-consensus-protocol etcd \  
      ↵
```

The first argument must be the cluster directory, for example, `speedy` or `~/clusters/speedy` (the cluster is named `speedy` in both cases). We recommend that you keep all your clusters in a common directory, for example, `~/clusters`. The next argument must be `--architecture` to select an architecture, followed by `--layout`.

The command creates a directory named `~/clusters/speedy` and generates a configuration file named `config.yml` that follows the layout of the *BDR-Always-ON* architecture and gold layout. You can use the `tpaexec info` command to see what values are supported for the configuration options based on what you specified when running the `configure` command.

Common configuration options

Other configuration options include:

Owner

Every cluster must be directly traceable to a person responsible for the provisioned resources.

By default, a cluster is tagged as being owned by the login name of the user running `tpaexec provision`. If this name does not identify a person (for example, `postgres`, `ec2-user`), you must specify `--owner SomeId` to set an identifiable owner.

You may use your initials, or “Firstname Lastname”, or anything else that identifies you uniquely.

Platform options

The default value for `--platform` is `aws`. It is the platform supported by the BDR-Always-ON architecture.

Specify `--region` to specify any existing AWS region that you have access to (and that permits the required number of instances to be created). The default region is `eu-west-1`.

Specify `--instance-type` with any valid instance type for AWS. The default is `t3.micro`.

Subnet selection

By default, each cluster is assigned a random /28 subnet under 10.33/16, but depending on the architecture, there may be one or more subnets, and each subnet may be anywhere between a /24 and a /29.

Specify `--subnet` to use a particular subnet. For example, `--subnet 192.0.2.128/27`.

Alternatively, specify `--subnet-pattern` to generate random subnets (as many as required by the architecture) matching the given pattern. For example, `--subnet-pattern 192.0.x.x`.

Disk space

Specify `--root-volume-size` to set the size of the root volume in GB. For example, `--root-volume-size 64`. The default is 16GB. (Depending on the image used to create instances, there may be a minimum size for the root volume.)

For architectures that support separate postgres and barman volumes:

Specify `--postgres-volume-size` to set the size of the Postgres volume in GB. The default is 16GB.

Specify `--barman-volume-size` to set the size of the Barman volume in GB. The default is 32GB.

Distribution

Specify `--os` or `--distribution` to specify the OS to be used on the cluster’s instances. The value is case-sensitive.

The selected platform determines which distributions are available and which one is used by default. For more details, see `tpaexec info platforms/<platformname>`.

In general, you can use “Debian”, “RedHat”, and “Ubuntu” to select TPA images that have Postgres and other software preinstalled (to reduce deployment times). To use stock distribution images instead, append “-minimal” to the value, for example, `--distribution Debian-minimal`.

2ndQuadrant repositories

By default, TPAexec installs the 2ndQuadrant public repository and adds on any product repositories that the architecture requires.

Specify `--2Q-repositories source/name/release ...` to specify the complete list of 2ndQuadrant repositories to install on each instance in addition to the 2ndQuadrant public repository.

If you do this, you must first export `TPA_2Q_SUBSCRIPTION_TOKEN=xxx` before you run `tpaexec`. You can get a subscription token from the

[EnterpriseDB customer portal](#) (**Support>Softwaresubscriptions>Add**).

Software versions

By default TPAexec uses the latest major version of Postgres. Specify `--postgres-version` to install an earlier supported major version.

By default, TPAexec always installs the latest version of every package. This is usually the desired behavior, but in some testing scenarios, it may be necessary to select specific package versions. For example,

```
- postgres-package-version 10.4-2.pgdg90+1
- repmgr-package-version 4.0.5-1.pgdg90+1
- barman-package-version 2.4-1.pgdg90+1
- pglogical-package-version 2.2.0*
- bdr-package-version 3.0.2*
- pgbouncer-package-version 1.8*
```

Specify `--extra-packages` or `--extra-postgres-packages` to install additional packages. The former lists packages to install along with system packages, while the latter lists packages to install later along with postgres packages. (If you mention packages that depend on Postgres in the former list, the installation fails because Postgres is not yet installed.) The arguments are passed on to the package manager for installation without any modifications.

The `--extra-optional-packages` option behaves like `--extra-packages`, but it is not an error if the named packages cannot be installed.

Hostnames

By default, `tpaexec` randomly selects as many hostnames as it needs from a pre-approved list of several dozen names. This should be enough for most clusters.

Specify `--hostnames-from` to select names from a different list (for example, if you need more names than are available in the canned list). The file must contain one hostname per line.

Specify `--hostnames-pattern` to restrict hostnames to those matching the `egrep`-syntax pattern. If you choose to do this, you must ensure that the pattern matches only valid hostnames (`[a-zA-Z0-9-]`) and finds a sufficient number thereof.

Locations

By default, `tpaexec configure` uses the names first, second, and so on for any locations used by the selected architecture.

Specify `--location-names` to provide more meaningful names for each location.

Enable Commit At Most Once

Specify `--enable-camo` to set the pair of BDR primary instances in each region to be each other's Commit At Most Once (CAMO) partners. See

Commit At Most Once (CAMO) for more information.

Provision

The `tpaexec provision` command creates instances and other resources required by the cluster. The details of the process depend on the architecture (for example, BDR-Always-ON) and platform (for example, AWS) that you selected while configuring the cluster.

For example, given AWS access with the necessary privileges, TPAexec provisions EC2 instances, VPCs, subnets, routing tables, internet gateways, security groups, EBS volumes, elastic IPs, and so on.

You can also “provision” existing servers by selecting the “bare” platform and providing connection details. Whether these are bare metal servers or those provisioned separately on a cloud platform, they can be used just as if they had been created by TPAexec.

You are not restricted to a single platform—you can spread your cluster out across some AWS instances (in multiple regions) and some on-premise servers, or servers in other data centres, as needed.

At the end of the provisioning stage, you will have the required number of instances with the basic operating system installed, which TPAexec can access via SSH (with `sudo` to root).

Deploy

The `tpaexec deploy` command installs and configures Postgres and other software on the provisioned servers (which may or may not have been created by TPAexec; but it doesn't matter who created them so long as SSH and `sudo` access is available). This includes setting up replication, backups, and so on.

At the end of the deployment stage, EDB Postgres Distributed is up and running.

Test

The `tpaexec test` command executes various architecture and platform-specific tests against the deployed cluster to ensure that it is working as expected.

At the end of the testing stage, you will have a fully-functioning cluster.

For more information, see [TPAexec](#).

9.1.5.3 Deploying EDB Postgres Distributed

The following steps setup EDB Postgres Distributed with the Always On Silverarchitecture using Amazon EC2.

1. Generate a configuration file:

```
$ tpaexec configure myedbdpcluster --architecture BDR-Always-ON --layout Silver --  
→platform aws --bdr-version 4 --2q --2Q-repositories products/2ndqpostgres/release
```

This creates a subdirectory directory in current working directory called `myedbdpcluster` containing the `config.yml` configuration file TPAexec uses to create the cluster. Edit the `config.yml` as needed, for example to change the IP address range used for servers or adjust locations of nodes.

We included options to specify using BDR 4 (the default version) and the 2ndQuadrant repositories to install EDB Postgres Extended Server. By default, TPAexec installs the 2ndQuadrant public repository and adds on any product repositories that the architecture requires. In this example, we specified `--2Q-repositories` so the complete list of 2ndQuadrant repositories is installed on each instance in addition to the 2ndQuadrant public repository. Since we did this, before the provisioning step, you must first export `TPA_2Q_SUBSCRIPTION_TOKEN=xxx`. You can get a subscription token from the [EnterpriseDB customer portal](#) (**Support>Softwaresubscriptions>Add**).

```
cd ~/clusters/myedbdpcluster  
  
export TPA_2Q_SUBSCRIPTION_TOKEN=<token>
```

1. Provision the cluster:

```
tpaexec provision myedbdpcluster
```

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

1. Deploy the needed packages, configuration and setup the actual EDB Postgres Distributed cluster:

```
tpaexec deploy myedbdpcluster
```

After the successful run of the `deploy` command the cluster is ready to use. You can connect to it via `psql` or any other database client.

It's also possible to run a test that ensures the cluster is running as expected:

```
tpaexec test myedbdpcluster
```

UPGRADING

Because EDB Postgres Distributed consists in multiple software components, the upgrade strategy depends partially on which components are being upgraded.

In general it's possible to upgrade the cluster with almost zero upgrade, by using an approach called Rolling Upgrade where nodes are upgraded one by one, and the application connections are switched over to already upgraded nodes.

It's also possible to stop all nodes, perform the upgrade on all nodes and only then restart the entire cluster, just like with a standard PostgreSQL setup. This strategy of upgrading all nodes at the same time avoids running with mixed versions of software and therefore is the simplest, but obviously incurs some downtime and is not recommended unless the Rolling Upgrade is not possible for some reason.

To upgrade an EDB Postgres Distributed cluster, perform the following steps:

1. Plan the upgrade.
1. Prepare for the upgrade.
1. Upgrade the server software.
1. Check and validate the upgrade.

10.1 Upgrade Planning

There are broadly two ways to upgrade each node.

- **Upgrading nodes in-place to the newer software version, see** [Rolling Server Software Upgrades](#) .
- **Replacing nodes with ones that have the newer version installed, see** [Rolling Upgrade Using Node Join](#) .

Both of these approaches can be done in a rolling manner.

10.1.1 Rolling Upgrade considerations

While the cluster is going through a rolling upgrade, mixed versions of software are running in the cluster. For example, nodeA has BDR 3.7.16, while nodeB and nodeC has 4.1.0. In this state, the replication and group management uses the protocol and features from the oldest version (3.7.16 in case of this example), so any new features provided by the newer version which require changes in the protocol are disabled. Once all nodes are upgraded to the same version, the new features are automatically enabled.

Similarly, when a cluster with WAL decoder enabled nodes is going through a rolling upgrade, WAL decoder on a higher version of BDR node produces LCRs with a higher pglogical version and WAL decoder on a lower version of BDR node produces LCRs with lower pglogical version. As a result, WAL senders on a higher version of BDR nodes are not expected to use LCRs due to a mismatch in protocol versions while on a lower version of BDR nodes, WAL senders may continue to use LCRs. Once all the BDR nodes are on the same BDR version, WAL senders use LCRs.

A rolling upgrade starts with a cluster with all nodes at a prior release, then proceeds by upgrading one node at a time to the newer release, until all nodes are at the newer release. There should never be more than two versions of any component running at the same time, which means the new upgrade must not be initiated until the previous upgrade process has fully finished on all nodes.

An upgrade process may take an extended period of time when the user decides caution is required to reduce business risk, though it's not recommended to run the mixed versions of the software indefinitely.

While Rolling Upgrade can be used for upgrading major version of the software it is not supported to mix PostgreSQL, EDB Postgres Extended and EDB Postgres Advanced Server in one cluster, so this approach cannot be used to change the Postgres variant.

Warning: Downgrades of the EDB Postgres Distributed are *not* supported and require manual rebuild of the cluster.

10.1.2 Rolling Server Software Upgrades

A rolling upgrade is the process where the *Server Software Upgrade* process is performed on each node in the cluster one after another, while keeping the remainder of the cluster operational.

The actual procedure depends on whether the Postgres component is being upgraded to a new major version or not.

During the upgrade process, the application can be switched over to a node which is currently not being upgraded to provide continuous availability of the database for applications.

10.1.3 Rolling Upgrade Using Node Join

The other method of upgrade of the server software, is to join a new node to the cluster and later drop one of the existing nodes running the older version of the software.

For this approach, the procedure is always the same, however because it includes node join, the potentially large data transfer is required.

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

Note: *bdr_init_physical* makes a byte-by-byte of the source node so it cannot be used while upgrading from one major Postgres version to another. In fact, currently *bdr_init_physical* requires that even the BDR version of the source and the joining node is exactly the same. It cannot be used for rolling upgrades via joining a new node method. Instead, a logical join must be used.

10.1.4 Upgrading a CAMO-Enabled Cluster

CAMO protection requires at least one of the nodes of a CAMO pair to be operational. For upgrades, we recommend to ensure that no CAMO protected transactions are running concurrent to the upgrade, or to use a rolling upgrade strategy, giving the nodes enough time to reconcile in between the upgrades and the corresponding node downtime due to the upgrade.

10.2 Upgrade Preparation

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

Please see individual changes mentioned in *Release notes* and any version-specific upgrade notes in this topic.

10.3 Server Software Upgrade

The upgrade of EDB Postgres Distributed on individual nodes happens in-place. There is no need for backup and restore when upgrading the BDR extension.

10.3.1 BDR Extension Upgrade

BDR extension upgrade process consists of few simple steps.

10.3.2 Stop Postgres

During the upgrade of binary packages, it's usually best to stop the running Postgres server first to ensure that mixed versions don't get loaded in case of unexpected restart during the upgrade.

10.3.3 Upgrade Packages

The first step in the upgrade is to install the new version of the BDR packages, which installs both the new binary and the extension SQL script. This step is operating system-specific.

10.3.4 Start Postgres

Once packages are upgraded the Postgres instance can be started, the BDR extension is automatically upgraded upon start when the new binaries detect older version of the extension.

10.3.5 Postgres Upgrade

The process of in-place upgrade of Postgres highly depends on whether you are upgrading to new minor version of Postgres or to new major version of Postgres.

10.3.6 Minor Version Postgres Upgrade

Upgrading to a new minor version of Postgres is similar to *upgrading the BDR extension*. Stopping Postgres, upgrading packages, and starting Postgres again is typically all that's needed.

However, sometimes additional steps like reindexing may be recommended for specific minor version upgrades. Refer to the Release Notes of the specific version of Postgres you are upgrading to.

10.3.7 Major Version Postgres Upgrade

Upgrading to a new major version of Postgres is a more complicated process.

EDB Postgres Distributed provides a `bdr_pg_upgrade` command line utility, which can be used to do a *In-place Postgres Major Version Upgrades*.

Note: When upgrading to new major version of any software, including Postgres, the BDR extension and others, it's always important to ensure the compatibility of your application with the target version of a given software.

10.4 Upgrade Check and Validation

After this procedure, your BDR node is upgraded. You can verify the current version of BDR4 binary like this:

```
SELECT bdr.bdr_version();
```

Always check the *Monitoring* after upgrade of a node to confirm that the upgraded node is working as expected.

10.5 Application Schema Upgrades

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

To eliminate this downtime, BDR offers ways to perform a rolling application schema upgrade.

10.5.1 Rolling Application Schema Upgrades

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

Warning: Rolling Application Schema Upgrades have to be managed outside of BDR. Careful scripting is required to make this work correctly on production clusters. Extensive testing is advised.

See *Replicating between nodes with differences* for details.

When one node runs DDL that adds a new table, nodes that have not yet received the latest DDL need to handle the extra table. In view of this, the appropriate setting for rolling schema upgrades is to configure all nodes to apply the `skip` resolver in case of a `target_table_missing` conflict. This must be performed before any node has additional tables added and is intended to be a permanent setting.

This is done with the following query, that must be **executed separately on each node**, after replacing `node1` with the actual node name:

```
SELECT bdr.alter_node_set_conflict_resolver(node1,
      target_table_missing, skip);
```

When one node runs DDL that adds a column to a table, nodes that have not yet received the latest DDL need to handle the extra columns. In view of this, the appropriate setting for rolling schema upgrades is to configure all nodes to apply the `ignore` resolver in case of a `target_column_missing` conflict. This must be performed before one node has additional columns added and is intended to be a permanent setting.

This is done with the following query, that must be **executed separately on each node**, after replacing `node1` with the actual node name:

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

When one node runs DDL that removes a column from a table, nodes that have not yet received the latest DDL need to handle the missing column. This situation will cause a `source_column_missing` conflict, which uses the `use_default_value` resolver. Thus, columns that neither accept NULLs nor have a DEFAULT value require a two step process:

1. Remove NOT NULL constraint or add a DEFAULT value for a column on all nodes.
2. Remove the column.

Constraints can be removed in a rolling manner. There is currently no supported way for handling adding table constraints in a rolling manner, one node at a time.

When one node runs a DDL that changes the type of an existing column, depending on the existence of binary coercibility between the current type and the target type, the operation may not rewrite the underlying table data. In that case, it will be only a metadata update of the underlying column type. Rewrite of a table is normally restricted. However, in controlled DBA environments, it is possible to change the type of a column to an automatically castable one by adopting a rolling upgrade for the type of this column in a non-replicated environment on all the nodes, one by one. See

`ALTER TABLE` for more details. section.

10.5.1.1 In-place Postgres Major Version Upgrades

Upgrading a BDR Node to a newer major version of Postgres is possible using the command-line utility `bdr_pg_upgrade`.

`bdr_pg_upgrade` internally uses the standard *`bdr_pg_upgrade command-line`* with BDR specific logic to ensure a smooth upgrade.

Terminology

Various terminology is used in this documentation to describe the upgrade process and components involved.

- old cluster* - The existing Postgres cluster node to be upgraded, which data will be migrated from.
- new cluster* - The new Postgres cluster, which data will be migrated to. This cluster node must be one (1) major version ahead of the old cluster.

Precautions

Standard Postgres major version upgrade precautions apply, including the fact that all the requirements for *`bdr_pg_upgrade command-line`* must be met by both clusters.

Additionally, `bdr_pg_upgrade` should not be used if there are other tools using replication slots and replication origins, only BDR slots and origins will be restored after the upgrade.

There are several prerequisites for `bdr_pg_upgrade` that have to be met:

- Applications using the old cluster have been disconnected, it can for example, be redirected to another node in the cluster
- Peer authentication is configured for both clusters, `bdr_pg_upgrade` requires peer authentication
- BDR versions on both clusters must be exactly the same and must be version 4.1.0 or above
- The new cluster must be in a shutdown state
- BDR packages must be installed in the new cluster
- The new cluster must be already initialized and configured as needed to match the old cluster configuration
- Databases, tables, and other objects must not exist in the new cluster

It is also recommended to have the old cluster up prior to running `bdr_pg_upgrade` as the CLI will start the old cluster if it is shutdown.

Usage

To upgrade to a newer major version of Postgres, the new version must first be installed.

`bdr_pg_upgrade` command-line

`bdr_pg_upgrade` passes all parameters to `pg_upgrade` . Therefore, you can specify any parameters supported by *`bdr_pg_upgrade` command-line* .

Synopsis

<code>bdr_pg_upgrade [OPTION] ...</code>
--

Options

In addition to the options for `pg_upgrade` , the following parameters can be passed to `bdr_pg_upgrade` :

- `-b, --old-bindir` - old cluster bin directory (required)
- `-B, --new-bindir` - new cluster bin directory (required)
- `-d, --old-datadir` - old cluster data directory (required)
- `-D, --new-datadir` - REQUIRED new cluster data directory (required)
- `--database` - BDR database name (required)
- `-p, --old-port` - old cluster port number
- `-s, --socketdir` - directory to use for postmaster sockets during upgrade
- `--check` - only perform checks, do not modify clusters

Environment Variables

Environment variables can be used in place of command line parameters.

- PGBINOLD - old cluster bin directory
- PGBINNEW - new cluster bin directory
- PGDATAOLD - old cluster data directory
- PGDATANEW - new cluster data directory
- PGPORTOLD - old cluster port number
- PGSOCKETDIR - directory to use for postmaster sockets during upgrade

Example

Given a scenario where:

- Old cluster bin directory is `/usr/lib/postgresql/13/bin`
- New cluster bin directory is `/usr/lib/postgresql/14/bin`
- Old cluster data directory is `/var/lib/postgresql/13/main`
- New cluster data directory is `/var/lib/postgresql/14/main`
- Database name is `bdrdb`

The following command could be used to upgrade the cluster:

```
bdr_pg_upgrade \
- -old-bindir /usr/lib/postgresql/13/bin \
- -new-bindir /usr/lib/postgresql/14/bin \
- -old-datadir /var/lib/postgresql/13/main \
- -new-datadir /var/lib/postgresql/14/main \
- -database bdrdb
```

Steps Performed

Steps performed when running `bdr_pg_upgrade`.

Note: When `-check` is supplied as an argument to `bdr_pg_upgrade`, the CLI will skip steps that modify the database.

BDR Postgres Checks

Steps	<code>-check</code> supplied
Collecting pre-upgrade new cluster control data	<i>run</i>
Checking new cluster state is shutdown	<i>run</i>
Checking BDR versions	<i>run</i>
Starting old cluster (if shutdown)	<i>skip</i>
Connecting to old cluster	<i>skip</i>

continues on next page

Table 1 – continued from previous page

Steps	<i>–check</i> supplied
Checking if bdr schema exists	<i>skip</i>
Turning DDL replication off	<i>skip</i>
Terminating connections to database.	<i>skip</i>
Disabling connections to database	<i>skip</i>
Waiting for all slots to be flushed	<i>skip</i>
Disconnecting from old cluster	<i>skip</i>
Stopping old cluster	<i>skip</i>
Starting old cluster with BDR disabled	<i>skip</i>
Connecting to old cluster	<i>skip</i>
Collecting replication origins	<i>skip</i>
Collecting replication slots	<i>skip</i>
Disconnecting from old cluster	<i>skip</i>
Stopping old cluster	<i>skip</i>

pg_upgrade Steps

Standard `pg_upgrade` steps are performed

Note: *–check* is passed to `pg_upgrade` if supplied

BDR Post-Upgrade Steps

Steps	<i>–check</i> supplied
Collecting old cluster control data	<i>skip</i>
Collecting new cluster control data	<i>skip</i>
Advancing LSN of new cluster	<i>skip</i>
Starting new cluster with BDR disabled	<i>skip</i>
Connecting to new cluster	<i>skip</i>
Creating replication origin Repeated for each origin	<i>skip</i>
Advancing replication origin Repeated for each origin	<i>skip</i>
Creating replication slot Repeated for each slot	<i>skip</i>
Stopping new cluster	<i>skip</i>

COMPATIBILITY MATRIX

BDR	PGD CLI	HARP	PostgreSQL	EDB Postgres Extended	EDB Postgres Advanced
4	1	2	12-14	12-14	12-14
3.7		2	11-13	11-13	11-13
3.6		2	10-11	11	

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

BACKUP AND RECOVERY

In this chapter we discuss the backup and restore of a BDR cluster.

BDR is designed to be a distributed, highly available system. If one or more nodes of a cluster are lost, the best way to replace them is to clone new nodes directly from the remaining nodes.

The role of backup and recovery in BDR is to provide for Disaster Recovery (DR), such as in the following situations:

- Loss of all nodes in the cluster
- Significant, uncorrectable data corruption across multiple nodes as a result of data corruption, application error or security breach

12.1 Backup

12.1.1 pg_dump

`pg_dump`, sometimes referred to as “logical backup”, can be used normally with BDR.

Note that `pg_dump` dumps both local and global sequences as if they were local sequences. This is intentional, to allow a BDR schema to be dumped and ported to other PostgreSQL databases. This means that sequence kind metadata is lost at the time of dump, so a restore would effectively reset all sequence kinds to the value of `bdr.default_sequence_kind` at time of restore.

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

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

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

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

12.1.2 Physical Backup

Physical backups of a node in a BDR cluster can be taken using standard PostgreSQL software, such as [Barman](#).

A physical backup of a BDR node can be performed with the same procedure that applies to any PostgreSQL node: a BDR node is just a PostgreSQL node running the BDR extension.

There are some specific points that must be considered when applying PostgreSQL backup techniques to BDR:

- BDR operates at the level of a single database, while a physical backup includes all the databases in the instance; you should plan your databases to allow them to be easily backed-up and restored.
- Backups will make a copy of just one node. In the simplest case, every node has a copy of all data, so you would need to backup only one node to capture all data. However, the goal of BDR will not be met if the site containing that single copy goes down, so the minimum should be at least one node backup per site (obviously with many copies etc.).
- However, each node may have un-replicated local data, and/or the definition of replication sets may be complex so that all nodes do not subscribe to all replication sets. In these cases, backup planning must also include plans for how to backup any unreplicated local data and a backup of at least one node that subscribes to each replication set.

12.1.3 Eventual Consistency

The nodes in a BDR cluster are *eventually consistent*, but not *entirely consistent*; a physical backup of a given node will provide Point-In-Time Recovery capabilities limited to the state actually assumed by that node (see the [Example] below).

The following example shows how two nodes in the same BDR cluster might not (and usually do not) go through the same sequence of states.

Consider a cluster with two nodes N1 and N2, which is initially in state S. If transaction W1 is applied to node N1, and at the same time a non-conflicting transaction W2 is applied to node N2, then node N1 will go through the following states:

(N1) S --> S + W1 --> S + W1 + W2
--

... while node N2 will go through the following states:

(N2) S --> S + W2 --> S + W1 + W2
--

That is: node N1 will *never* assume state S + W2, and node N2 likewise will never assume state S + W1, but both nodes will end up in the same state S + W1 + W2. Considering this situation might affect how you decide upon your backup strategy.

12.1.4 Point-In-Time Recovery (PITR)

In the example above, the changes are also inconsistent in time, since W1 and W2 both occur at time T1, but the change W1 is not applied to N2 until T2.

PostgreSQL PITR is designed around the assumption of changes arriving from a single master in COMMIT order. Thus, PITR is possible by simply scanning through changes until one particular point-in-time (PIT) is reached. With this scheme, you can restore one node to a single point-in-time from its viewpoint, e.g. T1, but that state would not include other data from other nodes that had committed near that time but had not yet arrived on the node. As a result, the recovery might be considered to be partially inconsistent, or at least consistent for only one replication origin.

To request this, use the standard syntax:

```
recovery_target_time = T1
```

BDR allows for changes from multiple masters, all recorded within the WAL log for one node, separately identified using replication origin identifiers.

BDR allows PITR of all or some replication origins to a specific point in time, providing a fully consistent viewpoint across all subsets of nodes.

Thus for multi-origins, we view the WAL stream as containing multiple streams all mixed up into one larger stream. There is still just one PIT, but that will be reached as different points for each origin separately.

We read the WAL stream until requested origins have found their PIT. We apply all changes up until that point, except that we do not mark as committed any transaction records for an origin after the PIT on that origin has been reached.

We end up with one LSN “stopping point” in WAL, but we also have one single timestamp applied consistently, just as we do with “single origin PITR”.

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

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

In the specific example above, N1 would be restored to T1, but would also include changes from other nodes that have been committed by T1, even though they were not applied on N1 until later.

To request multi-origin PITR, use the standard syntax in the `recovery.conf` file:

```
recovery_target_time = T1
```

The list of replication origins which would be restored to T1 need either to be specified in a separate `multi_recovery.conf` file via the use of a new parameter `recovery_target_origins`:

```
recovery_target_origins = *
```

...or one can specify the origin subset as a list in `recovery_target_origins`.

```
recovery_target_origins = 1,3
```

Note that the local WAL activity recovery to the specified `recovery_target_time` is always performed implicitly. For origins that are not specified in `recovery_target_origins`, recovery may stop at any point, depending on when the target for the list mentioned in `recovery_target_origins` is achieved.

In the absence of the `multi_recovery.conf` file, the recovery defaults to the original PostgreSQL PITR behaviour that is designed around the assumption of changes arriving from a single master in COMMIT order.

Note: This feature is only available on EDB Postgres Extended and Barman does not currently automatically create a `multi_recovery.conf` file.

12.2 Restore

While you can take a physical backup with the same procedure as a standard PostgreSQL node, what is slightly more complex is **restoring** the physical backup of a BDR node.

12.2.1 BDR Cluster Failure or Seeding a New Cluster from a Backup

The most common use case for restoring a physical backup involves the failure or replacement of all the BDR nodes in a cluster, for instance in the event of a datacentre failure.

You may also want to perform this procedure to clone the current contents of a BDR cluster to seed a QA or development instance.

In that case, BDR capabilities can be restored based on a physical backup of a single BDR node, optionally plus WAL archives:

- If you still have some BDR nodes live and running, fence off the host you restored the BDR node to, so it cannot connect to any surviving BDR nodes. This ensures that the new node does not confuse the existing cluster.
- Restore a single PostgreSQL node from a physical backup of one of the BDR nodes.
- If you have WAL archives associated with the backup, create a suitable `recovery.conf` and start PostgreSQL in recovery to replay up to the latest state. You can specify a alternative `recovery_target` here if needed.
- Start the restored node, or promote it to read/write if it was in standby recovery. Keep it fenced from any surviving nodes!
- Clean up any leftover BDR metadata that was included in the physical backup, as described below.
- Fully stop and restart the PostgreSQL instance.
- Add further BDR nodes with the standard procedure based on the `bdr.join_node_group()` function call.

12.2.2 Cleanup BDR Metadata

The cleaning of leftover BDR metadata is achieved as follows:

1. Drop the BDR node using `bdr.drop_node`
2. Fully stop and re-start PostgreSQL (important!).

12.2.3 Cleanup of Replication Origins

Replication origins must be explicitly removed with a separate step because they are recorded persistently in a system catalog, and therefore included in the backup and in the restored instance. They are not removed automatically when dropping the BDR extension, because they are not explicitly recorded as its dependencies.

BDR creates one replication origin for each remote master node, to track progress of incoming replication in a crash-safe way. Therefore we need to run:

```
SELECT pg_replication_origin_drop(bdr_dbname_grpname_nodename);
```

...once for each node in the (previous) cluster. Replication origins can be listed as follows:

```
SELECT * FROM pg_replication_origin;
```

...and those created by BDR are easily recognized by their name, as in the example shown above.

12.2.4 Cleanup of Replication Slots

If a physical backup was created with `pg_basebackup`, replication slots will be omitted from the backup.

Some other backup methods may preserve replication slots, likely in outdated or invalid states. Once you restore the backup, just:

```
SELECT pg_drop_replication_slot(slot_name)
FROM pg_replication_slots;
```

...to drop *all* replication slots. If you have a reason to preserve some, you can add a `WHERE slot_name LIKE 'bdr%'` clause, but this is rarely useful.

Warning: Never run this on a live BDR node.

MONITORING

Monitoring replication setups is important to ensure that your system performs optimally and does not run out of disk space or encounter other faults that may halt operations.

It is important to have automated monitoring in place to ensure that if, for example, replication slots start falling badly behind, the administrator is alerted and can take proactive action.

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

13.1 Monitoring Overview

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

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

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

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

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

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

13.2 Monitoring Node Join and Removal

By default, the node management functions wait for the join or partoperation to complete. This can be turned off using the respective `wait_for_completion` function argument. If waiting is turned off, then to see when a join or part operation finishes, check the node state indirectly via `bdr.node_summary` and `bdr.state_journal_details`.

When called, the helper function `bdr.wait_for_join_completion()` will cause a PostgreSQL session to pause until all outstanding node join operations complete.

Here is an example output of a `SELECT` query from `bdr.node_summary` that indicates that two nodes are active and another one is joining:

```
# SELECT node_name, interface_connstr, peer_state_name,
#        node_seq_id, node_local_dbname
# FROM bdr.node_summary;
- [ RECORD 1 ]-----+-----
node_name      | node1
interface_connstr | host=localhost dbname=postgres port=7432
peer_state_name | ACTIVE
node_seq_id    | 1
node_local_dbname | postgres
- [ RECORD 2 ]-----+-----
node_name      | node2
interface_connstr | host=localhost dbname=postgres port=7433
peer_state_name | ACTIVE
node_seq_id    | 2
node_local_dbname | postgres
- [ RECORD 3 ]-----+-----
node_name      | node3
interface_connstr | host=localhost dbname=postgres port=7434
peer_state_name | JOINING
node_seq_id    | 3
node_local_dbname | postgres
```

Also, the table `bdr.node_catchup_info` will give information on the catch-up state, which can be relevant to joining nodes or parting nodes.

When a node is parted, it could be that some nodes in the cluster did not receive all the data from that parting node. So it will create a temporary slot from a node that already received that data and can forward it.

The `catchup_state` can be one of the following:

```
10 = setup
20 = start
30 = catchup
40 = done
```

13.3 Monitoring Replication Peers

There are two main views used for monitoring of replication activity:

- `bdr.node_slots` for monitoring outgoing replication
- `bdr.subscription_summary` for monitoring incoming replication

Most of the information provided by `bdr.node_slots` can be also obtained by querying the standard PostgreSQL replication monitoring views

`pg_catalog.pg_stat_replication` and `pg_catalog.pg_replication_slots`.

Each node has one BDR group slot which should never have a connection to it and will very rarely be marked as active. This is normal, and does not imply something is down or disconnected. See Replication Slots created by BDR.

13.3.1 Monitoring Outgoing Replication

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

- `bdr.node_replication_rates` for monitoring outgoing replication

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

```
# SELECT * FROM bdr.node_replication_rates;
- [ RECORD 1 ]-----+-----
peer_node_id      | 112898766
target_name       | node1
sent_lsn          | 0/28AF99C8
replay_lsn        | 0/28AF99C8
replay_lag        | 00:00:00
replay_lag_bytes  | 0
replay_lag_size   | 0 bytes
apply_rate        | 822
catchup_interval  | 00:00:00
- [ RECORD 2 ]-----+-----
peer_node_id      | 312494765
target_name       | node3
sent_lsn          | 0/28AF99C8
replay_lsn        | 0/28AF99C8
replay_lag        | 00:00:00
replay_lag_bytes  | 0
replay_lag_size   | 0 bytes
apply_rate        | 853
catchup_interval  | 00:00:00
```

The `apply_rate` above refers to the rate in bytes per second. It is the rate at which the peer is consuming data from the local node. The `replay_lag` when a node reconnects to the cluster is immediately set to zero. We are working on fixing this information; as a workaround, we suggest you use the `catchup_interval` column that refers to the time required for the peer node to catch up to the local node data. The other fields are also available via the `bdr.node_slots` view, as explained below.

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

Administrators may query `bdr.node_slots` for outgoing replication from the local node. It shows information about replication status of all other nodes in the group that are known to the current node, as well as any additional replication slots created by BDR on the current node.

```
# SELECT node_group_name, target_dbname, target_name, slot_name, active_pid,
#        catalog_xmin, client_addr, sent_lsn, replay_lsn, replay_lag,
#        replay_lag_bytes, replay_lag_size
# FROM bdr.node_slots;
```

```
- [ RECORD 1 ]-----
```

node_group_name	bdrgroup
target_dbname	postgres
target_name	node3
slot_name	bdr_postgres_bdrgroup_node3
active_pid	15089
catalog_xmin	691
client_addr	127.0.0.1
sent_lsn	0/23F7B70
replay_lsn	0/23F7B70
replay_lag	[NULL]
replay_lag_bytes	120
replay_lag_size	120 bytes

```
- [ RECORD 2 ]-----
```

node_group_name	bdrgroup
target_dbname	postgres
target_name	node2
slot_name	bdr_postgres_bdrgroup_node2
active_pid	15031
catalog_xmin	691
client_addr	127.0.0.1
sent_lsn	0/23F7B70
replay_lsn	0/23F7B70
replay_lag	[NULL]
replay_lag_bytes	84211
replay_lag_size	82 kB

Note that because BDR is a mesh network, to get full view of lag in the cluster, this query has to be executed on all nodes participating.

`replay_lag_bytes` reports the difference in WAL positions between the local server's current WAL write position and `replay_lsn`, the last position confirmed replayed by the peer node. `replay_lag_size` is just a human-readable form of the same. It is important to understand that WAL usually contains a lot of writes that are not replicated but still count in `replay_lag_bytes`, including VACUUM activity, index changes, writes associated with other databases on the same node, writes for tables that are not part of a replication set, etc. So the lag in bytes reported here is not the amount of data that must be replicated on the wire to bring the peer node up to date, only the amount of server-side WAL that must be processed.

Similarly, `replay_lag` is not a measure of how long the peer node will take to catch up, or how long it will take to replay from its current position to the write position at the time `bdr.node_slots` was queried. It measures the delay between when the peer confirmed the most recent commit and the current wall-clock time. We suggest that you monitor `replay_lag_bytes` and `replay_lag_size` or `catchup_interval` in `bdr.node_replication_rates`, as this column is set to zero immediately after the node reconnects.

The lag in both bytes and time does not advance while logical replication is streaming a transaction. It only changes when a commit is replicated. So the lag will tend to "sawtooth", rising as a transaction is streamed, then falling again as the peer node commits it, flushes it, and sends confirmation. The reported LSN positions will "stair-step" instead of advancing smoothly, for similar reasons.

When replication is disconnected (`active = 'f'`), the `active_pid` column will be NULL, as will `client_addr` and other fields that only make sense with an active connection. The `state` field will be 'disconnected'. The `_lsn` fields will be the same as the `confirmed_flush_lsn`, since that is the last position that the client is known for certain to have replayed to and saved. The `_lag` fields will show the elapsed time between the most recent confirmed flush on the client and the current time, and the `_lag_size` and `_lag_bytes` fields will report the distance between `confirmed_flush_lsn` and the local server's current WAL insert position.

Note: It is normal for `restart_lsn` to be behind the other `lsn` columns; this does not indicate a problem with replication or a peer node lagging. The `restart_lsn` is the position that PostgreSQL's internal logical decoding must be reading WAL at if interrupted, and generally reflects the position of the oldest transaction that is not yet replicated and flushed. A very old `restart_lsn` can make replication slow to restart after disconnection and force retention of more WAL than is desirable, but will otherwise be harmless. If you are concerned, look for very long running transactions and forgotten prepared transactions.

13.3.2 Monitoring Incoming Replication

Incoming replication (also called subscription) can be monitored by querying the `bdr.subscription_summary` view. This shows the list of known subscriptions to other nodes in the BDR cluster and the state of the replication worker, e.g.:

```
# SELECT node_group_name, origin_name, sub_enabled, sub_slot_name,
#        subscription_status
# FROM bdr.subscription_summary;
- [ RECORD 1 ]-----+-----
node_group_name | bdrgroup
origin_name     | node2
sub_enabled     | t
sub_slot_name   | bdr_postgres_bdrgroup_node1
subscription_status | replicating
- [ RECORD 2 ]-----+-----
node_group_name | bdrgroup
origin_name     | node3
sub_enabled     | t
sub_slot_name   | bdr_postgres_bdrgroup_node1
subscription_status | replicating
```

13.3.3 Monitoring WAL senders using LCR

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

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

If `is_using_lcr` is `FALSE`, `decoder_slot_name` / `lcr_file_name` will be `NULL`. This will be the case if the Decoding Worker is not enabled, or the WAL sender is serving a

logical standby.

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

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

13.4 Monitoring BDR Replication Workers

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

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

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

The view `bdr.worker_errors` shows errors (if any) reported by any worker which has a problem continuing the work. Only active errors are visible in this view, so if the worker was having transient problems but has recovered, the view will be empty.

13.5 Monitoring BDR Writers

There is another system view `bdr.writers` to monitor writer activities. This view shows the current status of only writer workers. It includes:

- `sub_name` to identify the subscription which the writer belongs to
- `pid` of the writer process
- `streaming_allowed` to know if the writer supports application of in-progress streaming transactions
- `is_streaming` to know if the writer is currently applying a streaming transaction
- `commit_queue_position` to check the position of the writer in the commit queue.

BDR honours commit ordering by following the same commit order as happened on the origin. In case of parallel writers, multiple writers could be applying different transactions at the same time. The `commit_queue_position` shows in which order they will commit. Value `0` means that the writer is the first one to commit. Value `-1` means that the commit position is not yet known. This can happen for a streaming transaction or when the writer is not applying any transaction at the moment.

13.6 Monitoring Global Locks

The global lock, which is currently only used for DDL replication, is a heavyweight lock that exists across the whole BDR group.

There are currently two types of global locks:

- DDL lock, used for serializing all DDL operations on permanent (not temporary) objects (i.e. tables) in the database
- DML relation lock, used for locking out writes to relations during DDL operations that change the relation definition

Either or both entry types may be created for the same transaction, depending on the type of DDL operation and the value of the `bdr.ddl_locking` setting.

Global locks held on the local node are visible in the `'bdr.global_locks'` view `</bdr/latest/catalogs#bdrglobal_locks>`_`. This view shows the type of the lock; for relation locks it shows which relation is being locked, the PID holding the lock (if local), and whether the lock has been globally granted or not. In case of global advisory locks, `lock_type` column shows `GLOBAL_LOCK_ADVISORY` and `relation` column shows the advisory key(s) on which the lock is acquired.

The following is an example output of `bdr.global_locks` while running an `ALTER TABLE` statement with `bdr.ddl_locking = on`:

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

See the catalog documentation for details on all fields including lock timing information.

13.7 Monitoring Conflicts

Replication *Monitoring Conflicts* can arise when multiple nodes make changes that affect the same rows in ways that can interact with each other. The BDR system should be monitored to ensure that conflicts are identified and, where possible, application changes are made to eliminate them or make them less frequent.

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

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

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

13.8 External Monitoring

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

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

13.8.1 `bdr.set_node_location`

This function inserts node metadata into `bdr.node_location`

13.8.2 Synopsis

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

13.8.3 Parameters

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

13.8.4 `bdr.set_network_path_info`

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

13.8.5 Synopsis

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

13.8.6 Parameters

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

13.8.7 `bdr.network_monitoring` view

This view collects information about the network path between nodes.

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

13.9 Apply Statistics

BDR collects statistics about replication apply, both for each subscription and for each table.

Two monitoring views exist: `bdr.stat_subscription` for subscription statistics and `bdr.stat_relation` for relation statistics. These views both provide:

- Number of INSERTs/UPDATEs/DELETEs/TRUNCATEs replicated
- Block accesses and cache hit ratio
- Total I/O time for read/write
- Number of in-progress transactions streamed to file
- Number of in-progress transactions streamed to writers
- Number of in-progress streamed transactions committed/aborted

and for relations only, these statistics:

- Total time spent processing replication for the relation
- Total lock wait time to acquire lock (if any) for the relation (only)

and for subscriptions only, these statistics:

- Number of COMMITs/DDl replicated for the subscription
- Number of times this subscription has connected upstream

Tracking of these statistics is controlled by the BDR GUCs `bdr.track_subscription_apply` and `bdr.track_relation_apply` respectively.

The example output from these would look like this:

```
# SELECT sub_name, nconnect, ninsert, ncommit, nupdate, ndelete, ntruncate, nddl
FROM bdr.stat_subscription;
- [ RECORD 1 ]-----
sub_name | bdr_regression_bdrgroup_node1_node2
nconnect | 3
ninsert  | 10
ncommit  | 5
nupdate  | 0
ndelete  | 0
ntruncate| 0
nddl     | 2
```

In this case the subscription connected 3 times to the upstream, inserted 10 rows and did 2 DDL commands inside 5 transactions.

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

13.10 Standard PostgreSQL Statistics Views

Statistics on table and index usage are updated normally by the downstream master. This is essential for the correct function of

`autovacuum`. If there are no local writes on the downstream master

and statistics have not been reset, these two views should show corresponding results between upstream and downstream:

- `pg_stat_user_tables`
- `pg_statio_user_tables`

Note: We don't necessarily expect the upstream table statistics to be *similar* to the downstream ones; we only expect them to *change* by the same amounts. Consider the example of a table whose statistics show 1M inserts and 1M updates; when a new node joins the BDR group, the statistics for the same table in the new node will show 1M inserts and zero updates. However, from that moment, the upstream and downstream table statistics will change by the same amounts, because all changes on one side will be replicated to the other side.

Since indexes are used to apply changes, the identifying indexes on the downstream side may appear more heavily used with workloads that perform `UPDATE`s and `DELETE`s than non-identifying indexes are.

The built-in index monitoring views are:

- `pg_stat_user_indexes`
- `pg_statio_user_indexes`

All these views are discussed in detail in the [PostgreSQL documentation on the statistics views](#).

13.11 Monitoring BDR Versions

BDR allows running different Postgres versions as well as different BDR versions across the nodes in the same cluster. This is useful for upgrading.

The view `bdr.group_versions_details` uses the function `bdr.run_on_all_nodes()` to retrieve Postgres and BDR versions from all nodes at the same time. For example:

```
bdrdb=# SELECT node_name, postgres_version, bdr_version
        FROM bdr.group_versions_details;
 node_name | postgres_version | bdr_version
-----+-----+-----
 node1     | 14.1             | 4.0.0
 node2     | 14.1             | 4.0.0
```

The recommended setup is to try to have all nodes running the same latest versions as soon as possible. It is recommended that the cluster does not run different BDR versions for too long.

For monitoring purposes, we recommend the following alert levels:

- status=UNKNOWN, message=This node is not part of any BDR group
- status=OK, message=All nodes are running same BDR versions
- status=WARNING, message=There is at least 1 node that is not accessible
- status=WARNING, message=There are node(s) running different BDR versions when compared to other nodes

The described behavior is implemented in the function `bdr.monitor_group_versions()`, which uses BDR version information returned from the view `bdr.group_version_details` to provide a cluster-wide version check. For example:

```
bdrdb=# SELECT * FROM bdr.monitor_group_versions();
 status | message
-----+-----
 OK     | All nodes are running same BDR versions
```

13.12 Monitoring Raft Consensus

Raft Consensus should be working cluster-wide at all times. The impact of running a BDR cluster without Raft Consensus working might be as follows:

- BDR data changes replication may still be working correctly
- Global DDL/DML locks will not work
- Galloc sequences will eventually run out of chunks
- Eager Replication will not work
- Cluster maintenance operations (join node, part node, promote standby) are still allowed but they might not finish (simply hang)
- Node statuses might not be correctly synced among the BDR nodes
- BDR group replication slot does not advance LSN, thus keeps WAL files on disk

The view `bdr.group_raft_details` uses the functions `bdr.run_on_all_nodes()` and `bdr.get_raft_status()` to retrieve Raft Consensus status from all nodes at the same time. For example:

```
bdrdb=# SELECT node_id, node_name, state, leader_id
FROM bdr.group_raft_details;
 node_id | node_name | state      | leader_id
-----+-----+-----+-----
 1148549230 | node1      | RAFT_LEADER | 1148549230
 3367056606 | node2      | RAFT_FOLLOWER | 1148549230
```

We can say that Raft Consensus is working correctly if all below conditions are met:

- A valid state (RAFT_LEADER or RAFT_FOLLOWER) is defined on all nodes
- Only one of the nodes is the RAFT_LEADER
- The leader_id is the same on all rows and must match the node_id of the row where state = RAFT_LEADER

From time to time, Raft Consensus will start a new election to define anew RAFT_LEADER . During an election, there might be an intermediary situation where there is no RAFT_LEADER and some of the nodes consider themselves as RAFT_CANDIDATE . The whole election should not take longer than bdr.raft_election_timeout (by default it is set to 6 seconds). If the query above returns an in-election situation, then simply wait for bdr.raft_election_timeout and run the query again. If after bdr.raft_election_timeout has passed and some the conditions above are still not met, then Raft Consensus is not working.

Raft Consensus might not be working correctly on a single node only; for example one of the nodes does not recognize the current leader and considers itself as a RAFT_CANDIDATE . In this case, it is important to make sure that:

- All BDR nodes are accessible to each other through both regular and replication connections (check file pg_hba.conf)
- BDR versions are the same on all nodes
- bdr.raft_election_timeout is the same on all nodes

In some cases, especially if nodes are geographically distant from each other and/or network latency is high, the default value of bdr.raft_election_timeout (6 seconds) might not be enough. If Raft Consensus is still not working even after making sure everything is correct, consider increasing bdr.raft_election_timeout to, say, 30 seconds on all nodes. From BDR 3.6.11 onwards, setting bdr.raft_election_timeout requires only a server reload.

Given how Raft Consensus affects cluster operational tasks, and also as Raft Consensus is directly responsible for advancing the group slot, we can define monitoring alert levels as follows:

- status=UNKNOWN, message=This node is not part of any BDR group
- status=OK, message=Raft Consensus is working correctly
- status=WARNING, message=There is at least 1 node that is not accessible
- status=WARNING, message=There are node(s) as RAFT_CANDIDATE, an election might be in progress
- status=WARNING, message=There is no RAFT_LEADER, an election might be in progress
- status=CRITICAL, message=There is a single node in Raft Consensus
- status=CRITICAL, message=There are node(s) as RAFT_CANDIDATE while a RAFT_LEADER is defined
- status=CRITICAL, message=There are node(s) following a leader different than the node set as RAFT_LEADER

The described behavior is implemented in the function bdr.monitor_group_raft() , which uses Raft Consensus status information returned from the view bdr.group_raft_details to provide a cluster-wide Raft check. For example:

```
bdrdb=# SELECT * FROM bdr.monitor_group_raft();
 status | message
```

(continues on next page)

(continued from previous page)

```
-----+-----
OK      | Raft Consensus is working correctly
```

13.13 Monitoring Replication Slots

Each BDR node keeps:

- One replication slot per active BDR peer
- One group replication slot

For example:

```
bdrdb=# SELECT slot_name, database, active, confirmed_flush_lsn
FROM pg_replication_slots ORDER BY slot_name;
      slot_name      | database | active | confirmed_flush_lsn
-----+-----+-----+-----
bdr_bdrdb_bdrgroup   | bdrdb   | f      | 0/3110A08
bdr_bdrdb_bdrgroup_node2 | bdrdb   | t      | 0/31F4670
bdr_bdrdb_bdrgroup_node3 | bdrdb   | t      | 0/31F4670
bdr_bdrdb_bdrgroup_node4 | bdrdb   | t      | 0/31F4670
```

Peer slot names follow the convention `bdr_<DATABASE>_<GROUP>_<PEER>`, while the BDR group slot name follows the convention `bdr_<DATABASE>_<GROUP>`, which can be accessed using the function `bdr.local_group_slot_name()`.

Peer replication slots should be active on all nodes at all times. If a peer replication slot is not active, then it might mean:

- The corresponding peer is shutdown or not accessible; or
- BDR replication is broken.

Grep the log file for ERROR or FATAL and also check `bdr.worker_errors` on all nodes. The root cause might be, for example, an incompatible DDL was executed with DDL replication disabled on one of the nodes.

The BDR group replication slot is however inactive most of the time. BDR maintains this slot and advances its LSN when all other peers have already consumed the corresponding transactions. Consequently it is not necessary to monitor the status of the group slot.

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

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

One of the following status summaries will be returned:

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

13.14 Monitoring Transaction COMMITs

By default, BDR transactions commit only on the local node. In that case, transaction COMMIT will be processed quickly.

BDR can be used with standard PostgreSQL synchronous replication, while BDR also provides two new transaction commit modes: CAMO and Eager replication. Each of these modes provides additional robustness features, though at the expense of additional latency at COMMIT. The additional time at COMMIT can be monitored dynamically using the `bdr.stat_activity` catalog, where processes report different `wait_event` states. A transaction in COMMIT waiting for confirmations from one or more synchronous standbys reports a `SyncRep` wait event, whereas the two new modes report `EagerRep`.

PGD CLI

The EDB Postgres Distributed Command Line Interface (PGD CLI) is a tool to manage your BDR cluster. It allows you to run commands against BDR clusters.

You can use it to inspect and manage cluster resources and to get information on the following:

- node summary
- BDR/Postgres version on each node
- replication and subscription details
- membership and worker events
- raft consensus details
- CAMO (Commit at Most Once) details
- clock skew
- cluster health summary

See EDB Postgres Distributed Command Line Interface for more information.