



BDR (Bi-Directional Replication)

Release 3.7.0

EnterpriseDB

Oct 18, 2021

CONTENTS

1	Architectural Overview	3
1.1	Basic Architecture	4
1.1.1	Multiple Groups	4
1.1.2	Multiple Masters	4
1.1.3	Asynchronous, by default	4
1.1.4	Mesh Topology	4
1.1.5	Logical Replication	4
1.1.6	High Availability	5
1.1.7	Limits	5
1.2	Architectural Options & Performance	5
1.2.1	Characterising BDR performance	5
1.2.2	BDR MultiMaster within one Group	6
1.2.3	BDR AlwaysOn	6
1.2.4	BDR Worldwide	6
1.2.5	BDR AutoScale	7
1.3	Deployment	8
1.4	Clocks and Timezones	8
2	Application Usage	9
2.1	Application Behavior	9
2.2	Non-replicated statements	11
2.3	DML and DDL Replication	11
2.4	Replicating between different release levels	11
2.5	Replicating between nodes with differences	12
2.6	Comparison between nodes with differences	13
2.7	General Rules for Applications	13
2.8	Timing Considerations and Synchronous Replication	14
2.9	Application Testing	14
2.9.1	TPAexec	14
2.9.2	pgbench with CAMO/Failover options	14
2.9.3	isolationtester with multi-node access	15
2.10	Performance Testing & Tuning	17
2.11	Assessing Suitability	18
2.11.1	Assessing updates of Primary Key/Replica Identity	18
2.11.2	Assessing use of LOCK on tables or in SELECT queries	18
3	PostgreSQL Configuration for BDR	21
3.1	PostgreSQL Settings for BDR	21
3.2	2ndQPostgres/EDB Postgres Extended Settings for BDR	22
3.3	pglogical Settings for BDR	22

3.4	BDR Specific Settings	23
3.4.1	Conflict Handling	23
3.4.2	Global Sequence Parameters	23
3.4.3	DDL Handling	23
3.4.4	Global Locking	24
3.4.5	Node Management	24
3.4.6	Generic Replication	24
3.4.7	CRDTs	25
3.4.8	Max Prepared Transactions	26
3.4.9	Eager Replication	26
3.4.10	Commit at Most Once	26
3.4.11	Timestamp-based Snapshots	27
3.4.12	Monitoring and Logging	27
3.4.13	Internals	27
4	Node Management	29
4.1	Creating and Joining a BDR Group	29
4.1.1	Joining a Heterogeneous Cluster	30
4.1.2	Connection DSNs and SSL (TLS)	31
4.2	Witness Nodes	31
4.3	Logical Standby Nodes	32
4.4	Physical Standby Nodes	33
4.5	Sub-Groups	35
4.5.1	Subscriber-Only Groups	35
4.6	Decoding Worker	36
4.6.1	Notes	36
4.7	Node Restart and Down Node Recovery	37
4.7.1	Replication Slots created by BDR	37
4.7.2	Hashing Long Identifiers	38
4.8	Removing a Node From a BDR Group	39
4.8.1	Uninstalling BDR	39
4.9	Listing BDR Topology	40
4.9.1	Listing BDR Groups	40
4.9.2	Listing Nodes in a BDR Group	40
4.9.3	List of Node States	41
4.10	Managing Shard Groups	41
4.10.1	Create/Drop	42
4.10.2	Expand	42
4.10.3	Re-Balance	42
4.11	Node Management Interfaces	42
4.11.1	bdr.create_node	42
4.11.2	bdr.drop_node	43
4.11.3	bdr.create_node_group	43
4.11.4	bdr.alter_node_group_config	44
4.11.5	bdr.join_node_group	45
4.11.6	bdr.promote_node	46
4.11.7	bdr.wait_for_join_completion	47
4.11.8	bdr.part_node	47
4.11.9	bdr.alter_node_interface	48
4.11.10	bdr.alter_subscription_enable	48
4.11.11	bdr.alter_subscription_disable	49
4.12	Node Management Commands	50
4.12.1	bdr_init_physical	50

5	DDL Replication	53
5.1	DDL Replication Options	53
5.2	Executing DDL on BDR Systems	54
5.3	DDL Locking Details	54
5.4	Minimizing the Impact of DDL	56
5.5	Handling DDL With Down Nodes	57
5.6	Statement Specific DDL Replication Concerns	57
5.7	BDR DDL Command Handling Matrix	58
5.7.1	ALTER SEQUENCE	62
5.7.2	ALTER TABLE	62
5.7.3	ALTER TYPE	66
5.7.4	COMMENT ON	66
5.7.5	CREATE SEQUENCE	66
5.7.6	CREATE TABLE	66
5.7.7	CREATE TABLE AS and SELECT INTO	66
5.7.8	EXPLAIN	66
5.7.9	GRANT and REVOKE	67
5.7.10	LOCK TABLE	67
5.7.11	SECURITY LABEL	67
5.7.12	TRUNCATE Replication	67
5.7.13	TRUNCATE Locking	67
5.7.14	Role manipulation statements	67
5.7.15	Restricted DDL Workarounds	68
5.7.16	BDR Functions that behave like DDL	70
6	Security and Roles	71
6.1	Catalog Tables	71
6.2	BDR Functions & Operators	71
6.3	Granting privileges on catalog objects	72
6.4	Role Management	72
6.5	Roles and Replication	72
6.6	Connection Role	73
6.7	Privilege Restrictions	73
6.7.1	Foreign Key Privileges	73
6.7.2	Triggers	73
6.8	BDR Default/Predefined Roles	74
6.8.1	bdr_superuser	74
6.8.2	bdr_read_all_stats	74
6.8.3	bdr_monitor	76
6.8.4	bdr_application	76
6.8.5	bdr_read_all_conflicts	77
6.9	Verification	77
6.9.1	Coverity	77
6.9.2	CIS Benchmark	77
7	Conflicts	79
7.1	How conflicts happen	80
7.2	Types of conflict	80
7.2.1	PRIMARY KEY or UNIQUE Conflicts	80
7.2.2	Foreign Key Constraint Conflicts	85
7.2.3	TRUNCATE Conflicts	87
7.2.4	Exclusion Constraint Conflicts	87
7.2.5	Data Conflicts for Roles and Tablespace differences	87
7.2.6	Lock Conflicts and Deadlock Aborts	88

7.2.7	Divergent Conflicts	88
7.2.8	TOAST Support Details	88
7.3	Avoiding or Tolerating Conflicts	89
7.4	Conflict Detection	90
7.4.1	Origin Conflict Detection	90
7.4.2	Row Version Conflict Detection	91
7.4.3	bdr.alter_table_conflict_detection	92
7.4.4	List of Conflict Types	93
7.5	Conflict Resolution	93
7.5.1	bdr.alter_node_set_conflict_resolver	93
7.5.2	List of Conflict Resolvers	94
7.5.3	Default Conflict Resolvers	96
7.5.4	List of Conflict Resolutions	96
7.6	Conflict Logging	96
7.6.1	bdr.alter_node_set_log_config	96
7.6.2	Conflict Reporting	98
7.7	Data Verification with LiveCompare	98
8	Sequences	99
8.1	BDR Global Sequences	99
8.1.1	Timeshard Sequences	100
8.1.2	Globally-allocated range Sequences	101
8.2	UUIDs, KSUUIDs and Other Approaches	103
8.2.1	UUIDs and KSUUIDs	103
8.2.2	Step & Offset Sequences	104
8.3	Global Sequence Management Interfaces	105
8.3.1	bdr.alter_sequence_set_kind	105
8.3.2	bdr.extract_timestamp_from_timeshard	106
8.3.3	bdr.extract_nodeid_from_timeshard	106
8.3.4	bdr.extract_localseqid_from_timeshard	106
8.3.5	bdr.timestamp_to_timeshard	107
8.4	KSUUID v2 Functions	107
8.4.1	bdr.gen_ksuuid_v2	107
8.4.2	bdr.ksuuid_v2_cmp	108
8.4.3	bdr.extract_timestamp_from_ksuuid_v2	108
8.5	KSUUID v1 Functions	109
8.5.1	bdr.gen_ksuuid	109
8.5.2	bdr.uuid_v1_cmp	109
8.5.3	bdr.extract_timestamp_from_ksuuid	110
9	Column-Level Conflict Detection	111
9.1	Enabling and Disabling Column-Level Conflict Resolution	112
9.1.1	Example	112
9.1.2	Listing Table with Column-Level Conflict Resolution	112
9.1.3	bdr.column_timestamps_create	113
9.2	DDL Locking	113
9.3	Current vs Commit Timestamp	113
9.4	Inspecting Column Timestamps	114
9.5	Handling column conflicts using CRDT Data Types	115
9.6	Limitations	115
10	Conflict-free Replicated Data Types	117
10.1	State-based and operation-based CRDTs	119
10.1.1	Operation-based CRDT Types (CmCRDT)	119

10.1.2	State-based CRDT Types (CvCRDT)	120
10.2	Disk-Space Requirements	120
10.3	CRDT Types vs Conflicts Handling	121
10.4	CRDT Types vs. Conflict Reporting	121
10.5	Resetting CRDT Values	121
10.6	Implemented CRDT data types	122
10.6.1	grow-only counter (<code>crdt_gcounter</code>)	122
10.6.2	grow-only sum (<code>crdt_gsum</code>)	123
10.6.3	positive-negative counter (<code>crdt_pncounter</code>)	124
10.6.4	positive-negative sum (<code>crdt_pnsum</code>)	125
10.6.5	delta counter (<code>crdt_delta_counter</code>)	126
10.6.6	delta sum (<code>crdt_delta_sum</code>)	127
11	Durability & Performance Options	129
12	Eager Replication	131
13	Commit At Most Once	133
14	Database Scaling	135
15	Timestamp-based Snapshots	137
16	Replication Sets	139
17	Stream Triggers	141
18	Backup and Recovery	143
19	Upgrades	145
20	Explicit Two-Phase Commit (2PC)	147
21	Catalogs and Views	149
22	Monitoring	151
22.1	Monitoring Overview	151
22.2	Monitoring Node Join and Removal	152
22.3	Monitoring Replication Peers	153
22.3.1	Monitoring Outgoing Replication	153
22.3.2	Monitoring Incoming Replication	155
22.3.3	Monitoring WAL senders using LCR	155
22.4	Monitoring BDR Replication Workers	156
22.5	Monitoring Global Locks	156
22.6	Monitoring Conflicts	157
22.7	External Monitoring	157
22.7.1	<code>bdr.set_node_location</code>	157
22.7.2	<code>bdr.set_network_path_info</code>	158
22.7.3	<code>bdr.network_monitoring</code> view	158
22.8	Apply Statistics	158
22.9	Standard PostgreSQL Statistics Views	159
22.10	Monitoring BDR Versions	160
22.11	Monitoring Raft Consensus	160
22.12	Monitoring Replication Slots	162
22.13	Monitoring Transaction COMMITS	163

23	BDR System Functions	165
23.1	Version Information Functions	165
23.1.1	bdr.bdr_edition	165
23.1.2	bdr.bdr_version	165
23.1.3	bdr.bdr_version_num	165
23.1.4	bdr.wal_sender_stats	166
23.2	System and Progress Information Parameters	166
23.2.1	bdr.local_node_id	166
23.2.2	bdr.last_committed_lsn	166
23.2.3	transaction_id	166
23.3	Utility Functions	167
23.3.1	bdr.wait_slot_confirm_lsn	167
23.3.2	bdr.wait_for_apply_queue	167
23.3.3	bdr.get_node_sub_receive_lsn	168
23.3.4	bdr.get_node_sub_apply_lsn	168
23.3.5	bdr.run_on_all_nodes	169
23.3.6	bdr.global_lock_table	170
23.3.7	bdr.wait_for_xid_progress	171
23.3.8	bdr.local_group_slot_name	172
23.3.9	bdr.node_group_type	172
23.4	Global Advisory Locks	172
23.4.1	bdr.global_advisory_lock	173
23.4.2	bdr.global_advisory_unlock	173
24	Credits and Licence	175
25	'Appendix A	177
25.1	BDR 3.7.12	177
25.1.1	Improvements	177
25.1.2	Resolved Issues	177
25.1.3	Upgrades	178
25.2	BDR 3.7.11	178
25.2.1	Improvements	178
25.2.2	Resolved Issues	179
25.2.3	Upgrades	179
25.3	BDR 3.7.10	179
25.3.1	Improvements	179
25.3.2	Resolved Issues	180
25.4	BDR 3.7.9	181
25.4.1	Improvements	181
25.4.2	Resolved Issues	181
25.5	BDR 3.7.8	182
25.5.1	Important Notes	182
25.5.2	The Highlights of BDR 3.7	183
25.5.3	Improvements	184
25.5.4	Resolved Issues	184
25.5.5	Other Changes	185
25.6	BDR 3.7.7	186
25.6.1	Important Notes	186
25.6.2	Improvements	186
25.6.3	Resolved Issues	187
25.7	BDR 3.7.6	188
25.7.1	Important Notes	188
25.7.2	Improvements	188

25.7.3	Resolved Issues	189
25.7.4	Other Changes	189
25.8	BDR 3.7.5	189
25.8.1	Important Notes	190
25.8.2	Improvements	190
25.8.3	Resolved Issues	191
25.9	BDR 3.7.4	191
25.9.1	Important Notes	191
25.9.2	Improvements	192
25.9.3	Resolved Issues	192
25.10	BDR 3.7.3	193
25.10.1	Important Notes	193
25.10.2	Improvements	194
25.10.3	Resolved Issues	194
25.10.4	Other Improvements	195
25.11	BDR 3.7.2	195
25.11.1	Important Notes	195
25.11.2	The Highlights of BDR 3.7	195
25.11.3	Other Improvements	196
25.11.4	Resolved Issues	196
25.12	BDR 3.6.19	197
25.12.1	Resolved Issues	197
25.12.2	Improvements	198
25.13	BDR 3.6.18	198
25.13.1	Improvements	198
25.13.2	Resolved Issues	199
25.14	BDR 3.6.17	199
25.14.1	Improvements	199
25.14.2	Resolved Issues	200
25.15	BDR 3.6.16	201
25.15.1	Improvements	201
25.15.2	Resolved Issues	201
25.16	BDR 3.6.15	202
25.16.1	Improvements	202
25.16.2	Resolved Issues	202
25.17	BDR 3.6.14	202
25.17.1	Improvements	203
25.17.2	Resolved Issues	203
25.18	BDR 3.6.12	204
25.18.1	Improvements	204
25.18.2	Resolved Issues	204
25.19	BDR 3.6.11	205
25.19.1	Improvements	205
25.19.2	Resolved Issues	206
25.20	BDR 3.6.10	207
25.20.1	Improvements	207
25.20.2	Resolved Issues	208
25.21	BDR 3.6.9	208
25.21.1	Improvements	208
25.21.2	Resolved Issues	208
25.22	BDR 3.6.8	208
25.22.1	Improvements	208
25.22.2	Resolved Issues	209
25.23	BDR 3.6.7.1	209

25.23.1 Resolved Issues	209
25.24 BDR 3.6.7	209
25.24.1 Improvements	210
25.24.2 Resolved Issues	210
25.25 BDR 3.6.6	211
25.25.1 Improvements	211
25.25.2 Resolved Issues	212
25.26 BDR 3.6.5	212
25.26.1 Improvements	212
25.26.2 Resolved Issues	213
25.27 BDR 3.6.4	213
25.27.1 The Highlights of BDR 3.6.4	214
25.27.2 Resolved Issues	214
25.27.3 Other Improvements	214
25.28 BDR 3.6.3	215
25.28.1 The Highlights of BDR 3.6.3	215
25.28.2 Resolved Issues	215
25.28.3 Other Improvements	215
25.29 BDR 3.6.2	216
25.29.1 The Highlights of BDR 3.6.2	216
25.29.2 Resolved Issues	216
25.30 BDR 3.6.1	217
25.30.1 The highlights of 3.6.1	217
25.30.2 Resolved Issues	217
25.31 BDR 3.6.0.2	218
25.31.1 Resolved Issues	218
25.32 BDR 3.6.0.1	218
25.32.1 Resolved Issues	218
25.33 BDR 3.6.0	218
25.33.1 The highlights of BDR 3.6	219
25.33.2 Resolved issues	219
25.33.3 Other improvements	219
26 'Appendix B	221
27 'Appendix C	223
28 'Appendix D	225
29 'Appendix E	227

title: BDR (Bi-Directional Replication)

originalFilePath: index.md

BDR (short for Bi-Directional Replication) is a PostgreSQL extension which provides a solution for building multi-master clusters with mesh topology. This means that you can write to any server and the changes will be sent row-by-row to all the other servers that are part of the same BDR group.

BDR version 3 (“BDR3”) is built on the [pglogical3](#) extension. However, everything you need to know about BDR3 is included here and it is unnecessary, as well as potentially confusing, to refer to pglogical docs.

This documentation refers only to BDR3, not to earlier architectures, referred to as BDR1 and BDR2. There are significant and important differences in BDR3 and you should not refer to earlier docs or rely on anything stated within them.

BDR3 supports different versions and variant of PostgreSQL, as shown below.

BDR version support compatibility matrix:

BDR	Server	Supported Versions
3.6	PostgreSQL	10, 11
3.6	EDB Postgres Extended	11
3.7	PostgreSQL	11, 12, 13
3.7	EDB Postgres Advanced	11, 12, 13
3.7	EDB Postgres Extended	11, 12, 13

EDB Postgres Extended was formerly known as 2ndQ Postgres.

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

- Conflict-free Replicated Data Types - additional data types which provide mathematically proven consistency in asynchronous multi-master update scenarios (EDB Postgres Advanced, EDB Postgres Extended)
- Column Level Conflict Resolution - ability to use per column last-update wins resolution so that UPDATES on different fields can be “merged” without losing either of them (EDB Postgres Advanced, EDB Postgres Extended)
- Transform Triggers - triggers that are executed on the incoming stream of data providing ability to modify it or to do advanced programmatic filtering (EDB Postgres Advanced, EDB Postgres Extended)
- Conflict triggers - triggers which are called when conflict is detected, providing a way to use custom conflict resolution techniques (EDB Postgres Advanced, EDB Postgres Extended)
- Additional DDL support (CREATE TABLE AS) (EDB Postgres Advanced, EDB Postgres Extended)
- Advanced DDL Handling for NOT VALID constraints and ALTER TABLE (EDB Postgres Advanced, EDB Postgres Extended)
- Additional synchronization for Logical/Physical Standby servers for faster build of failoverable standbys (EDB Postgres Advanced, EDB Postgres Extended)
- Parallel Apply - allow multiple writers to apply the incoming changes (EDB Postgres Advanced, EDB Postgres Extended)
- Eager Replication - synchronizes between the nodes of the cluster before committing a transaction to provide conflict free replication (EDB Postgres Extended)
- Commit At Most Once - a consistency feature helping an application to commit each transaction only once, even in the presence of node failures (EDB Postgres Extended)

- Timestamp-based Snapshots - providing consistent reads across multiple nodes for retrieving data as they appeared or will appear at a given time (EDB Postgres Extended)
- Estimates for Replication Catch-up times (EDB Postgres Extended)
- Hold back freezing to assist resolution of UPDATE/DELETE conflicts (EDB Postgres Extended)
- Decoding Worker (EDB Postgres Extended 13 and above)

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

ARCHITECTURAL OVERVIEW

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



Fig. 1: node diagram

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

1.1 Basic Architecture

1.1.1 Multiple Groups

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

1.1.2 Multiple Masters

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

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

You don't have to write to all the masters, all of the time; it's a frequent configuration to direct writes mostly to just one master. However, if you just want one-way replication, the use of `pglogical` may be more appropriate.

1.1.3 Asynchronous, by default

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

1.1.4 Mesh Topology

BDR is structured around a mesh network where every node connects to every other node and all nodes exchange data directly with each other. There is no forwarding of data within BDR except in special circumstances such as node addition and node removal. Data may arrive from outside the BDR cluster or be sent onwards using `pglogical` or native PostgreSQL logical replication.

1.1.5 Logical Replication

Logical replication is a method of replicating data rows and their changes, based upon their replication identity (usually a primary key). We use the term *logical* in contrast to *physical* replication, which uses exact block addresses and byte-by-byte replication. Index changes are not replicated, thereby avoiding write amplification and reducing bandwidth.

Logical replication starts by copying a snapshot of the data from the source node. Once that is done, later commits are sent to other nodes as they occur in real time. Changes are replicated without re-executing SQL, so the exact data written is replicated quickly and accurately.

Nodes apply data in the order in which commits were made on the source node, ensuring transactional consistency is guaranteed for the changes from any single node. Changes from different nodes are applied independently of other nodes to ensure the rapid replication of changes.

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

1.1.6 High Availability

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

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

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

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

1.1.7 Limits

BDR has been tested with up to 99 master nodes in one cluster, but it is currently designed for use with up to 32 master nodes. Each master node can be protected by multiple physical or logical standby nodes; there is no specific limit on the number of standby nodes, but typical usage would be to have 2-3 standbys per master, with a typical maximum of 32 standbys per master.

BDR assumes there will be no more than 1024 nodes (counting both master nodes and logical standbys for the total) when using timeshard sequences, not counting nodes that have been previously removed (parted/dropped) from a group.

BDR places a limit that at most 10 databases in any one PostgreSQL instance can be BDR nodes across different BDR node groups. Having multiple nodes/databases within one instance be part of the same BDR node group is not supported.

1.2 Architectural Options & Performance

1.2.1 Characterising BDR performance

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

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

The following architectures are available:

- Multimaster/Single Group
- BDR AlwaysOn
- BDR Worldwide
- BDR AutoScale

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

1.2.2 BDR MultiMaster within one Group

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

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

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

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

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

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

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

1.2.3 BDR AlwaysOn

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

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

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

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

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

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

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

1.2.4 BDR Worldwide

In this architecture, multiple BDR Groups exist in multiple worldwide Regions, but all nodes in a Group are always within one Region.

If high volume tables are configured to that they only replicate within their own Group, then peak write TPS will scale linearly according to the number of Groups. This allows for sharding data based upon its input location, also known as geo-sharding.

This makes this architecture very suitable for use in IoT, monitoring or other high volume data collection applications, where location is the way that the application is naturally partitioned. High volume data doesn't leave its Region,

allowing us to avoid high bandwidth costs from cross-region replication as well as allowing us to follow strong laws on data privacy and data jurisdiction.

BDR allows multi-node read queries in this architecture. Large multi-node queries will scale linearly in terms of response time as the number of nodes increases. Throughput is limited since each query runs on one node in every sub-cluster.

1.2.5 BDR AutoScale

In this architecture, we use an array of Groups to create a parallel compute platform.

By default, tables are created on all nodes. For range partitioned tables, the user can specify a distribution key that allows partitions to be striped across the Groups to allow automatic sharding. E.g. TimeStamp, CustomerId or WarehouseId. Note that individual key values may also be used instead of ranges, effectively supporting list-based partitioning.

Tables that are identical on all nodes are referred to as Replicated Tables. Tables that are sharded across groups are referred to as Distributed or Sharded Tables. Replicated tables can be joined to any other table. Distributed tables can only be joined to Replicated tables, or to other Distributed tables that share the exact same data distribution.

Distributed tables may not always be perfectly evenly distributed. If there are heavily accessed shards, we can add additional nodes to that Group to deal with the imbalanced workload.

This form of sharding allows OLTP write performance to scale linearly, if the application sends transactions directly to the appropriate group, which is known as Direct Routing. In this mode, the architecture can be used somewhat similarly to distributed NoSQL systems such as Apache Cassandra. Autoscale doesn't support hash partitioning because it causes problems when the cluster needs to expand or contract, limiting the ability to scale elastically as needed.

If the workload can't send transactions directly to the correct group, we execute them against a Coordinator node which then routes them to any node within the appropriate Group. The Coordinator is then acting as a proxy to perform Indirect or Proxy Routing. Each Group will have a preferred write node and 1-2 other shadow nodes; reads will be directed to the shadow nodes by default, or the preferred write node if it is the only one remaining.

The Coordinator node adds latency and can limit scalability as the fraction of time spent in the coordinator increases. Workloads with lots of small read only requests cause a higher % of coordinator time, whereas large decision support queries show a small % of coordinator time, so scale the best.

Shard Routing requires access to the Shard Distribution Metadata, held within BDR catalog tables. Coordinator nodes store this information in their catalog tables, so have direct local access. The Shard Distribution Metadata can also be cached within client-side programs or in routing middleware to allow Direct Routing to take place from those components.

AutoScale allows multi-node read queries in this architecture. Large multi-node queries will scale linearly in terms of response time as the number of nodes increases. Throughput is limited since each query runs on one node in every sub-cluster.

AutoScale doesn't yet support multi-node write transactions when using a Coordinator, but these will be supported in future releases. For now, these operations are accepted but are not atomic. Multi-node write transactions would limit scalability in a sharded architecture.

1.3 Deployment

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

Manual deployment is not recommended and may not be supported.

Please refer to the `TPAexec Architecture User Manual` for your architecture.

Log messages and documentation are currently available only in English.

1.4 Clocks and Timezones

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

Server clocks should be synchronized using NTP or other solutions.

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

APPLICATION USAGE

This chapter looks at BDR from an application or user perspective.

Setting up nodes is discussed in a later chapter, as is replication of DDL, and various options for controlling replication using replication sets.

2.1 Application Behavior

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

BDR will, by default, replicate all changes from INSERTs, UPDATEs, DELETEs and TRUNCATEs from the source node to other nodes. Only the final changes will be sent, after all triggers and rules have been processed. For example, `INSERT ... ON CONFLICT UPDATE` will send either an INSERT or an UPDATE depending on what occurred on the origin. If an UPDATE or DELETE affects zero rows, then no changes will be sent.

INSERTs can be replicated without any pre-conditions.

For UPDATEs and DELETEs to be replicated on other nodes, we must be able to identify the unique rows affected. BDR requires that a table have either a PRIMARY KEY defined, a UNIQUE constraint or have an explicit REPLICATION IDENTITY defined on specific column(s). If one of those is not defined, a WARNING will be generated, and later UPDATEs or DELETEs will be explicitly blocked. If REPLICATION IDENTITY FULL is defined for a table, then a unique index is not required; in that case, UPDATEs and DELETEs are allowed and will use the first non-unique index that is live, valid, not deferred and does not have expressions or WHERE clauses, otherwise a sequential scan will be used.

TRUNCATE can be used even without a defined replication identity. Replication of TRUNCATE commands is supported, but some care must be taken when truncating groups of tables connected by foreign keys. When replicating a truncate action, the subscriber will truncate the same group of tables that was truncated on the origin, either explicitly specified or implicitly collected via CASCADE, except in cases where replication sets are defined, see [Replication Sets](#) chapter for further details and examples. This will work correctly if all affected tables are part of the same subscription. But if some tables to be truncated on the subscriber have foreign-key links to tables that are not part of the same (or any) replication set, then the application of the truncate action on the subscriber will fail.

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

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

Sequences need special handling, described in the [Sequences](#) chapter.

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

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

Replication is only possible from base tables to base tables. That is, the tables on the source and target on the subscription side must be tables, not views, materialized views, or foreign tables. Attempts to replicate tables other than base tables will result in an error. DML changes that are made through updatable views are resolved through to base tables on the origin and then applied to the same base table name on the target.

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

By default, triggers execute only on the origin node. For example, an INSERT trigger executes on the origin node and is ignored when we apply the change on the target node. You can specify that triggers should execute on both the origin node at execution time and on the target when it is replicated (“apply time”) by using `ALTER TABLE ... ENABLE ALWAYS TRIGGER`, or use the `REPLICA` option to execute only at apply time, `ALTER TABLE ... ENABLE REPLICA TRIGGER`.

Some types of trigger are not executed on apply, even if they exist on a table and are currently enabled. Trigger types not executed are

- Statement-level triggers (`FOR EACH STATEMENT`)
- Per-column UPDATE triggers (`UPDATE OF column_name [, ...]`)

BDR replication apply uses the system-level default `search_path`. Replica triggers, stream triggers and index expression functions may assume other `search_path` settings which will then fail when they execute on apply. To ensure this does not occur, resolve object references clearly using either the default `search_path` only, always use fully qualified references to objects, e.g. `schema.objectname`, or set the search path for a function using `ALTER FUNCTION ... SET search_path = ...` for the functions affected.

Note that BDR assumes that there are no issues related to text or other collatable datatypes, i.e. all collations in use are available on all nodes and the default collation is the same on all nodes. Replication of changes uses equality searches to locate Replica Identity values, so this will not have any effect except where unique indexes are explicitly defined with non-matching collation qualifiers. Row filters might be affected by differences in collations if collatable expressions were used.

BDR handling of very-long “toasted” data within PostgreSQL is transparent to the user. Note that the TOAST “chunkid” values will likely differ between the same row on different nodes, but that does not cause any problems.

BDR cannot work correctly if Replica Identity columns are marked as “external”.

PostgreSQL allows `CHECK()` constraints that contain volatile functions. Since BDR re-executes `CHECK()` constraints on apply, any subsequent re-execution that doesn’t return the same result as previously will cause data divergence.

BDR does not restrict the use of Foreign Keys; cascading FKs are allowed.

BDR does not currently support the use of non-ASCII schema or relation names. Later versions will remove this restriction.

2.2 Non-replicated statements

None of the following user commands are replicated by BDR, so their effects occur on the local/origin node only:

- Cursor operations (DECLARE, CLOSE, FETCH)
- Execution commands (DO, CALL, PREPARE, EXECUTE, EXPLAIN)
- Session management (DEALLOCATE, DISCARD, LOAD)
- Parameter commands (SET, SHOW)
- Constraint manipulation (SET CONSTRAINTS)
- Locking commands (LOCK)
- Table Maintenance commands (VACUUM, ANALYZE, CLUSTER, REINDEX)
- Async operations (NOTIFY, LISTEN, UNLISTEN)

Note that since the NOTIFY SQL command and the `pg_notify()` functions are not replicated, notifications are *not* reliable in case of failover. This means that notifications could easily be lost at failover if a transaction is committed just at the point the server crashes. Applications running LISTEN may miss notifications in case of failover. This is regrettably true in standard PostgreSQL replication and BDR does not yet improve on this. CAMO and Eager replication options do not allow the NOTIFY SQL command or the `pg_notify()` function.

2.3 DML and DDL Replication

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

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

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

2.4 Replicating between different release levels

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

BDR is also designed to replicate between nodes that have different versions of BDR software. This is a feature designed to allow version upgrades and maintenance without downtime.

However, while it's possible to join a node with a major version in a cluster, you can not add a node with a minor version if the cluster uses a newer protocol version, this will return error.

Both of the above features may be affected by specific restrictions; any known incompatibilities will be described in the release notes.

2.5 Replicating between nodes with differences

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

Currently, replication requires the same table name on all nodes. A future feature may allow a mapping between different table names.

It is possible to replicate between tables with dissimilar partitioning definitions, such as a source which is a normal table replicating to a partitioned table, including support for updates that change partitions on the target. It can be faster if the partitioning definition is the same on the source and target since dynamic partition routing need not be executed at apply time. Further details are available in the chapter on Replication Sets.

By default, all columns are replicated. BDR replicates data columns based on the column name. If a column has the same name but a different datatype, we attempt to cast from the source type to the target type, if casts have been defined that allow that.

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

If the target has missing column(s) from the source then BDR will raise a `target_column_missing` conflict, for which the default conflict resolver is `ignore_if_null`. This will throw an `ERROR` if a non-NULL value arrives. Alternatively, a node can also be configured with a conflict resolver of `ignore`. This setting will not throw an `ERROR`, just silently ignore any additional columns.

If the target has additional column(s) not seen in the source record then BDR will raise a `source_column_missing` conflict, for which the default conflict resolver is `use_default_value`. Replication will proceed if the additional columns have a default, either `NULL` (if nullable) or a default expression, but will throw an `ERROR` and halt replication if not.

Transform triggers can also be used on tables to provide default values or alter the incoming data in various ways before apply.

If the source and the target have different constraints, then replication will be attempted, but it might fail if the rows from source cannot be applied to the target. Row filters may help here.

Replicating data from one schema to a more relaxed schema won't cause failures. Replicating data from a schema to a more restrictive schema will be a source of potential failures. The right way to solve this is to place a constraint on the more relaxed side, so bad data is prevented from being entered. That way, no bad data ever arrives via replication, so it will never fail the transform into the more restrictive schema. For example, if one schema has a column of type `TEXT` and another schema defines the same column as `XML`, add a `CHECK` constraint onto the `TEXT` column that enforces that the text is `XML`.

A table may be defined with different indexes on each node. By default, the index definitions will be replicated. Refer to [DDL Replication](#) to specify how to create an index only on a subset of nodes, or just locally.

Storage parameters, such as `fillfactor` and `toast_tuple_target`, may differ between nodes for a table without problems. An exception to that is the value of a table's storage parameter `user_catalog_table` must be identical on all nodes.

A table being replicated should be owned by the same user/role on each node. Refer to [Security and Roles](#) for further discussion.

Roles may have different passwords for connection on each node, though by default changes to roles are replicated to each node. Refer to [DDL Replication](#) to specify how to alter a role password only on a subset of nodes, or just locally.

2.6 Comparison between nodes with differences

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

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

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

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

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

2.7 General Rules for Applications

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

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

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

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

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

2.8 Timing Considerations and Synchronous Replication

Being asynchronous by default, peer nodes may lag behind making it's possible for a client connected to multiple BDR nodes or switching between them to read stale data.

A [queue wait function](#) is provided for clients or proxies to prevent such stale reads.

The synchronous replication features of Postgres are available to BDR as well. In addition, BDR provides multiple variants for more synchronous replication. Please refer to the [Durability & Performance Options](#) chapter for an overview and comparison of all variants available and its different modes.

2.9 Application Testing

BDR applications can be tested using the following programs, in addition to other techniques.

- *TPAexec*
- *pgbench with CAMO/Failover options*
- *isolationtester with multi-node access*

2.9.1 TPAexec

TPAexec is the system used by EDB to deploy reference TPA architectures, including those based on Postgres-BDR.

TPAexec includes test suites for each reference architecture; it also simplifies creating and managing a local collection of tests to be run against a TPA cluster, using a syntax as in the following example:

```
tpaexec test mycluster mytest
```

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

2.9.2 pgbench with CAMO/Failover options

pgbench has been extended to allow users to run failover tests while using CAMO or regular BDR deployments. The following new options have been added:

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

--retry
retry transactions on failover
```

in addition to the above options, the connection information about the peer node for failover must be specified in [DSN form](#).

- Use `-m camo` or `-m failover` to specify the mode for pgbench. The `-m failover` specification can be used to test failover in regular BDR deployments.
- Use `--retry` to specify whether transactions should be retried when failover happens with `-m failover` mode. This is enabled by default for `-m camo` mode.

Here's an example invocation in a CAMO environment:

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

The above command will run in `camo` mode. It will connect to `node1` and run the tests; if the connection to `node1` connection is lost, then `pgbench` will connect to `node2`. It will query `node2` to get the status of in-flight transactions. Aborted and in-flight transactions will be retried in `camo` mode.

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

2.9.3 isolationtester with multi-node access

`isolationtester` has been extended to allow users to run tests on multiple sessions and on multiple nodes. This is used for internal BDR testing, though it is also available for use with user application testing.

```
$ isolationtester \
  --outputdir=./iso_output \
  --create-role=logical \
  --dbname=postgres \
  --server 'd1=dbname=node1' \
  --server 'd2=dbname=node2' \
  --server 'd3=dbname=node3'
```

Isolation tests are a set of tests run for examining concurrent behaviors in PostgreSQL. These tests require running multiple interacting transactions, which requires management of multiple concurrent connections, and therefore can't be tested using the normal `pg_regress` program. The name “isolation” comes from the fact that the original motivation was to test the serializable isolation level; but tests for other sorts of concurrent behaviors have been added as well.

It is built using PGXS as an external module. On installation, it creates `isolationtester` binary file which is run by `pg_isolation_regress` to perform concurrent regression tests and observe results.

`pg_isolation_regress` is a tool similar to `pg_regress`, but instead of using `psql` to execute a test, it uses `isolationtester`. It accepts all the same command-line arguments as `pg_regress`. It has been modified to accept multiple hosts as parameters. It then passes these host conninfo's along with server names to `isolationtester` binary. `Isolationtester` compares these server names with the names specified in each session in the spec files and runs given tests on respective servers.

To define tests with overlapping transactions, we use test specification files with a custom syntax, which is described in the next section. To add a new test, place a spec file in the `specs/` subdirectory, add the expected output in the `expected/` subdirectory, and add the test's name to the `Makefile`.

`Isolationtester` is a program that uses `libpq` to open multiple connections, and executes a test specified by a spec file. A `libpq` connection string specifies the server and database to connect to; defaults derived from environment variables are used otherwise.

Specification consists of five parts, tested in this order:

```
server "<name>"
```

This defines the name of the servers that the sessions will run on. There can be zero or more server `<name>` specifications. The conninfo corresponding to the names is provided via the command to run `isolationtester`. This is described in `quickstart_isolationtest.md`. This part is optional.

```
setup { <SQL> }
```

The given SQL block is executed once, in one session only, before running the test. Create any test tables or other required objects here. This part is optional. Multiple setup blocks are allowed if needed; each is run separately, in the

given order. (The reason for allowing multiple setup blocks is that each block is run as a single PQexec submission, and some statements such as VACUUM cannot be combined with others in such a block.)

```
teardown { <SQL> }
```

The teardown SQL block is executed once after the test is finished. Use this to clean up in preparation for the next permutation, e.g dropping any test tables created by setup. This part is optional.

```
session "<name>"
```

There are normally several “session” parts in a spec file. Each session is executed in its own connection. A session part consists of three parts: setup, teardown and one or more “steps”. The per-session setup and teardown parts have the same syntax as the per-test setup and teardown described above, but they are executed in each session. The setup part typically contains a “BEGIN” command to begin a transaction.

Additionally, a session part also consists of `connect_to` specification. This points to server name specified in the beginning which indicates the server on which this session runs.

```
connect_to "<name>"
```

Each step has the syntax

```
step "<name>" { <SQL> }
```

where `<name>` is a name identifying this step, and `SQL` is a SQL statement (or statements, separated by semicolons) that is executed in the step. Step names must be unique across the whole spec file.

```
permutation "<step name>"
```

A permutation line specifies a list of steps that are run in that order. Any number of permutation lines can appear. If no permutation lines are given, the test program automatically generates all possible orderings of the steps from each session (running the steps of any one session in order). Note that the list of steps in a manually specified “permutation” line doesn’t actually have to be a permutation of the available steps; it could for instance repeat some steps more than once, or leave others out.

Lines beginning with a # are considered comments.

For each permutation of the session steps (whether these are manually specified in the spec file, or automatically generated), the isolation tester runs the main setup part, then per-session setup parts, then the selected session steps, then per-session teardown, then the main teardown script. Each selected step is sent to the connection associated with its session.

To run isolation tests in a BDR3 environment that ran all prerequisite make commands, follow the below steps,

1. Run `make isolationcheck-install` to install the isolationtester submodule
2. You can run isolation regression tests using either of the following commands from the bdr-private repo

```
make isolationcheck-installcheck make isolationcheck-makecheck
```

A. To run `isolationcheck-installcheck`, you need to have two or more postgresql servers running. Pass the conninfo’s of servers to `pg_isolation_regress` in BDR 3.0 Makefile. Ex: `pg_isolation_regress --server 'd1=host=myhost dbname=mydb port=5434' --server 'd2=host=myhost1 dbname=mydb port=5432'`

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

Then run `make isolationcheck-installcheck`

B. `Isolationcheck-makecheck` currently supports running isolation tests on a single instance by setting up BDR between multiple databases.

You need to pass appropriate database names, conninfos of bdr instances to `pg_isolation_regress` in BDR Makefile as follows: `pg_isolation_regress --dbname=db1,db2 --server 'd1=dbname=db1' --server 'd2=dbname=db2'`

Then run `make isolationcheck-makecheck`

Each step may contain commands that block until further action has been taken (most likely, some other session runs a step that unblocks it or causes a deadlock). A test that uses this ability must manually specify valid permutations, i.e. those that would not expect a blocked session to execute a command. If a test fails to follow that rule, `isolationtester` will cancel it after 300 seconds. If the cancel doesn't work, `isolationtester` will exit uncleanly after a total of 375 seconds of wait time. Testing invalid permutations should be avoided because they can make the isolation tests take a very long time to run, and they serve no useful testing purpose.

Note that `isolationtester` recognizes that a command has blocked by looking to see if it is shown as waiting in the `pg_locks` view; therefore, only blocks on heavyweight locks will be detected.

2.10 Performance Testing & Tuning

BDR allows you to issue write transactions onto multiple master nodes. Bringing those writes back together onto each node has a cost in performance that you should be aware of.

First, replaying changes from another node has a CPU cost, an I/O cost and it will generate WAL records. The resource usage is usually less than in the original transaction since CPU overheads are lower as a result of not needing to re-execute SQL. In the case of UPDATE and DELETE transactions there may be I/O costs on replay if data isn't cached.

Second, replaying changes holds table-level and row-level locks that can produce contention against local workloads. The CRDT (Conflict-free Replicated Data Types) and CLCD (Column-Level Conflict Detection) features ensure you get the correct answers even for concurrent updates, but they don't remove the normal locking overheads. If you get locking contention, try to avoid conflicting updates and/or keep transactions as short as possible. A heavily updated row within a larger transaction will cause a bottleneck on performance for that transaction. Complex applications require some thought to maintain scalability.

If you think you're having performance problems, you're encouraged to develop performance tests using the benchmarking tools above. `pgbench` allows you to write custom test scripts specific to your use case so you can understand the overheads of your SQL and measure the impact of concurrent execution.

So if "BDR is running slow", then we suggest the following:

1. Write a custom test script for `pgbench`, as close as you can make it to the production system's problem case.
2. Run the script on one node to give you a baseline figure.
3. Run the script on as many nodes as occurs in production, using the same number of sessions in total as you did on one node. This will show you the effect of moving to multiple nodes.
4. Increase the number of sessions for the above 2 tests, so you can plot the effect of increased contention on your application.
5. Make sure your tests are long enough to account for replication delays.
6. Ensure that replication delay isn't growing during your tests.

Use all of the normal Postgres tuning features to improve the speed of critical parts of your application.

2.11 Assessing Suitability

BDR is compatible with PostgreSQL, but not all PostgreSQL applications are suitable for use on distributed databases. Most applications are already, or can be easily modified to become BDR compliant. Users can undertake an assessment activity in which they can point their application to a BDR-enabled setup. BDR provides a few knobs which can be set during the assessment period. These will aid in the process of deciding suitability of their application in a BDR-enabled environment.

2.11.1 Assessing updates of Primary Key/Replica Identity

BDR cannot currently perform conflict resolution where the PRIMARY KEY is changed by an UPDATE operation. It is permissible to update the primary key, but you must ensure that no conflict with existing values is possible.

When running on EDB Postgres Extended, BDR provides the following configuration parameter to assess how frequently the primary key/replica identity of any table is being subjected to update operations.

Note that these configuration parameters must only be used for assessment only. They can be used on a single node BDR instance, but must not be used on a production BDR cluster with two or more nodes replicating to each other. In fact, a node may fail to start or a new node will fail to join the cluster if any of the assessment parameters are set to anything other than IGNORE.

`bdr.assess_update_replica_identity = IGNORE (default) | LOG | WARNING | ERROR`

By enabling this parameter during the assessment period, one can log updates to the key/replica identity values of a row. One can also potentially block such updates, if desired. E.g.

```
CREATE TABLE public.test(g int primary key, h int);
INSERT INTO test VALUES (1, 1);

SET bdr.assess_update_replica_identity TO 'error';
UPDATE test SET g = 4 WHERE g = 1;
ERROR:  bdr_assess: update of key/replica identity of table public.test
```

Apply worker processes will always ignore any settings for this parameter.

2.11.2 Assessing use of LOCK on tables or in SELECT queries

Because BDR writer processes operate much like normal user sessions, they are subject to the usual rules around row and table locking. This can sometimes lead to BDR writer processes waiting on locks held by user transactions, or even by each other.

When running on EDB Postgres Extended, BDR provides the following configuration parameter to assess if the application is taking explicit locks.

`bdr.assess_lock_statement = IGNORE (default) | LOG | WARNING | ERROR`

Two types of locks that can be tracked are:

- explicit table-level locking (LOCK TABLE ...) by user sessions
- explicit row-level locking (SELECT ... FOR UPDATE/FOR SHARE) by user sessions

By enabling this parameter during the assessment period, one can track (or block) such explicit locking activity. E.g.

```
CREATE TABLE public.test(g int primary key, h int);
INSERT INTO test VALUES (1, 1);

SET bdr.assess_lock_statement TO 'error';
SELECT * FROM test FOR UPDATE;
ERROR:  bdr_assess: "SELECT FOR UPDATE" invoked on a BDR node

SELECT * FROM test FOR SHARE;
ERROR:  bdr_assess: "SELECT FOR SHARE" invoked on a BDR node

SET bdr.assess_lock_statement TO 'warning';
LOCK TABLE test IN ACCESS SHARE MODE;
WARNING: bdr_assess: "LOCK STATEMENT" invoked on a BDR node
```


POSTGRES SQL CONFIGURATION FOR BDR

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

3.1 PostgreSQL Settings for BDR

BDR requires these PostgreSQL settings to run correctly:

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

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

- `logical_decoding_work_mem` - memory buffer size used by logical decoding. Transactions larger than this will overflow the buffer and be stored temporarily on local disk. Default 64MB, but can be set much higher.
- `max_worker_processes` - BDR uses background workers for replication and maintenance tasks, so there need to be enough worker slots for it to work correctly. The formula for the correct minimal number of workers is: one per PostgreSQL instance + one per database on that instance + four per BDR-enabled database + one per peer node in the BDR group + one for each writer enabled per peer node in the BDR group, for each database. Additional worker processes may be needed temporarily when node is being removed from a BDR group.
- `max_wal_senders` - Two needed per every peer node.
- `max_replication_slots` - Same as `max_wal_senders`.
- `wal_sender_timeout` and `wal_receiver_timeout` - Determine how quickly an origin considers its CAMO partner as disconnected or reconnected; see [CAMO Failure Scenarios](#) for details.

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

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

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

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

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

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

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

3.2 2ndQPostgres/EDB Postgres Extended Settings for BDR

The following Postgres settings need to be considered for commit at most once (CAMO), a feature that is only available for BDR in combination with 2ndQPostgres. Some of these are only available in 2ndQPostgres; others already exist in the community version, but only become relevant with BDR in combination with CAMO.

- `synchronous_replication_availability` - Can optionally be `async` for increased availability by allowing an origin to continue and commit after its CAMO partner got disconnected. Under the default value of `wait`, the origin will wait indefinitely, and proceed to commit only after the CAMO partner reconnects and sends confirmation.
- `snapshot_timestamp` - Turns on the usage of [timestamp-based snapshots](#) and sets the timestamp to use.

3.3 pglogical Settings for BDR

BDR is also affected by some of the `pglogical` settings as it uses `pglogical` internally to implement the basic replication.

- `pglogical.track_subscription_apply` - Track apply statistics for each subscription.
- `pglogical.track_relation_apply` - Track apply statistics for each relation.
- `pglogical.track_apply_lock_timing` - Track lock timing when tracking statistics for relations.
- `pglogical.standby_slot_names` - When using physical Standby nodes intended for failover purposes, should be set to the replication slot(s) for each intended Standby.
- `pglogical.writers_per_subscription` - Default number of writers per subscription (in BDR this can also be changed by `bdr.alter_node_group_config` for a group).
- `pglogical.max_writers_per_subscription` - Maximum number of writers per subscription (sets upper limit for the setting above).

3.4 BDR Specific Settings

There are also BDR specific configuration settings that can be set. Unless noted otherwise, values may be set by any user at any time.

3.4.1 Conflict Handling

- `bdr.default_conflict_detection` - Sets the default conflict detection method for newly created tables; accepts same values as `bdr.alter_table_conflict_detection()`

3.4.2 Global Sequence Parameters

- `bdr.default_sequence_kind` - Sets the default [sequence kind](#).

3.4.3 DDL Handling

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

The accepted values are:

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

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

BDR can not replicate `UPDATES` and `DELETES` on tables without a `PRIMARY KEY` or `UNIQUE` constraint, unless the replica identity for the table is `FULL`, either by table-specific configuration or via `bdr.default_replica_identity`.

If `bdr.default_replica_identity` is `DEFAULT` and there is a `UNIQUE` constraint on the table, it will not be automatically picked up as `REPLICA IDENTITY`. It needs to be set explicitly at the time of creating the table, or afterwards as described in the documentation above.

Setting the replica identity of table(s) to `FULL` increases the volume of WAL written and the amount of data replicated on the wire for the table.

- `bdr.ddl_replication` - Automatically replicate DDL across nodes (default “on”).

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

Running DDL or calling BDR administration functions with `bdr.ddl_replication = off` can create situations where replication stops until an administrator can intervene. See [the DDL replication chapter](#) for details.

A LOG-level log message is emitted to the PostgreSQL server logs whenever `bdr.ddl_replication` is set to `off`. Additionally, a `WARNING`-level message is written whenever replication of captured DDL commands or BDR replication functions is skipped due to this setting.

- `bdr.role_replication` - Automatically replicate `ROLE` commands across nodes (default “on”). This parameter is settable by a `superuser` only. This setting only works if `bdr.ddl_replication` is turned on as well.

Turning this off without using external methods to ensure roles are in sync across all nodes may cause replicated DDL to interrupt replication until the administrator intervenes.

See [Role manipulation statements in the DDL replication chapter](#) for details.

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

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

Possible options are:

- `off` - do not use global locking for DDL operations
- `on` - use global locking for all DDL operations
- `dml` - only use global locking for DDL operations that need to prevent writes by taking the global DML lock for a relation

A LOG-level log message is emitted to the PostgreSQL server logs whenever `bdr.ddl_replication` is set to `off`. Additionally, a **WARNING** message is written whenever any global locking steps are skipped due to this setting. It is normal for some statements to result in two **WARNINGS**, one for skipping the DML lock and one for skipping the DDL lock.

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

3.4.4 Global Locking

- `bdr.ddl_locking` - Described above.
- `bdr.global_lock_max_locks` - Maximum number of global locks that can be held on a node (default 1000). May only be set at Postgres server start.
- `bdr.global_lock_timeout` - Sets the maximum allowed duration of any wait for a global lock (default 1 minute). A value of zero disables this timeout.
- `bdr.global_lock_statement_timeout` - Sets the maximum allowed duration of any statement holding a global lock (default 10 minutes). A value of zero disables this timeout.
- `bdr.global_lock_idle_timeout` - Sets the maximum allowed duration of idle time in transaction holding a global lock (default 10 minutes). A value of zero disables this timeout.

3.4.5 Node Management

- `bdr.replay_progress_frequency` - Interval for sending replication position info to the rest of the cluster (default 1 minute).

3.4.6 Generic Replication

- `bdr.xact_replication` - Replicate current transaction (default "on").

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

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

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

!!! Note Even with transaction replication disabled, WAL will be generated but those changes will be filtered away on the origin.

!!! Warning Turning off `bdr.xact_replication` *will* lead to data inconsistency between nodes, and should only be used to recover from data divergence between nodes or in replication situations where changes on single nodes are required for replication to continue. Use at your own risk.

- `bdr.permit_unsafe_commands` - Option to override safety check on commands that are deemed unsafe for general use.

Requires `bdr_superuser` or PostgreSQL superuser.

!!! Warning The commands that are normally not considered safe may either produce inconsistent results or break replication altogether. Use at your own risk.

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

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

- `bdr.maximum_clock_skew`

This specifies what should be considered as the maximum difference between the incoming transaction commit timestamp and the current time on the subscriber before triggering `bdr.maximum_clock_skew_action`.

This checks if the timestamp of the currently replayed transaction is in the future compared to the current time on the subscriber; and if it is, and the difference is larger than `bdr.maximum_clock_skew`, it will do the action specified by the `bdr.maximum_clock_skew_action` setting.

The default is -1, which means: ignore clock skew (the check is turned off). It is valid to set 0 as when the clock on all servers are synchronized, the fact that we are replaying the transaction means it has been committed in the past.

- `bdr.maximum_clock_skew_action`

This specifies the action to take if a clock skew higher than `bdr.maximum_clock_skew` is detected.

There are two possible values for this option:

- **WARN** - Log a warning about this fact. The warnings are logged once per minute (the default) at the maximum to prevent flooding the server log.
- **WAIT** - Wait for as long as the current local timestamp is no longer older than remote commit timestamp minus the `bdr.maximum_clock_skew`.

3.4.7 CRDTs

- `bdr.crdt_raw_value` - Sets the output format of [CRDT Data Types](#). The default output (when this setting is off) is to return only the current value of the base CRDT type (for example, a bigint for `crdt_pncounter`). When set to on, the returned value represents the full representation of the CRDT value, which can for example include the state from multiple nodes.

3.4.8 Max Prepared Transactions

- `max_prepared_transactions` - Needs to be set high enough to cope with the maximum number of concurrent prepared transactions across the cluster due to explicit two-phase commits, CAMO or Eager transactions. Exceeding the limit prevents a node from running a local two-phase commit or CAMO transaction, and will prevent all Eager transactions on the cluster. May only be set at Postgres server start. (EDB Postgres Extended)

3.4.9 Eager Replication

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

!!! Note This is only available on EDB Postgres Extended.

3.4.10 Commit at Most Once

- `bdr.enable_camo` - Used to enable and control the CAMO feature. Defaults to `off`. CAMO can be switched on per transaction by setting this to `remote_write`, `remote_commit_async`, or `remote_commit_flush`. For backwards-compatibility, the values `on`, `true`, and `1` set the safest `remote_commit_flush` mode. While `false` or `0` also disable CAMO.
- `bdr.camo_partner_of` - Allows specifying a CAMO partner per database. Expects pairs of database name and node name joined by a colon. Multiple pairs may be specified, but only the first occurrence per database is taken into account. For example: `'db1:node_4 test_db:test_node_3'`. May only be set at Postgres server start.
- `bdr.camo_origin_for` - Per-database node name of the origin of transactions in a CAMO pairing; for each database, this needs to match with the `bdr.camo_partner_of` setting on the corresponding origin node. May only be set at Postgres server start.
- `bdr.standby_dsn` - Allows manual override of the connection string (DSN) to reach the CAMO partner, in case it has changed since the crash of the local node. Should usually be unset. May only be set at Postgres server start.
- `bdr.camo_local_mode_delay` - The commit delay that applies in CAMO's Local mode to emulate the overhead that normally occurs with the CAMO partner having to confirm transactions. Defaults to 5 ms. Setting to 0 disables this feature.
- `bdr.camo_enable_client_warnings` - Emit warnings if an activity is carried out in the database for which CAMO properties cannot be guaranteed. This is enabled by default. Well-informed users can choose to disable this to reduce the amount of warnings going into their logs.

!!! Note This is only available on EDB Postgres Extended.

3.4.11 Timestamp-based Snapshots

- `bdr.timestamp_snapshot_keep` - For how long to keep valid snapshots for the timestamp-based snapshot usage (default 0, meaning do not keep past snapshots). Also see `snapshot_timestamp` above. (EDB Postgres Extended)

3.4.12 Monitoring and Logging

- `bdr.debug_level` - Defines the log level that BDR uses to write its debug messages. The default value is `debug2`. If you want to see detailed BDR debug output, set `bdr.debug_level = 'log'`.
- `bdr.trace_level` - Similar to the above, this defines the log level to use for BDR trace messages. Enabling tracing on all nodes of a BDR cluster may help 2ndQuadrant Support to diagnose issues. May only be set at Postgres server start.

!!! Warning Setting `bdr.debug_level` or `bdr.trace_level` to a value $\geq$ `log_min_messages` can produce a very large volume of log output, so it should not be enabled long term in production unless plans are in place for log filtering, archival and rotation to prevent disk space exhaustion.

3.4.13 Internals

- `bdr.raft_keep_min_entries` - The minimum number of entries to keep in the Raft log when doing log compaction (default 100). The value of 0 will disable log compaction. **WARNING: If log compaction is disabled, the log will grow in size forever.** May only be set at Postgres server start.
- `bdr.raft_response_timeout` - To account for network failures, the Raft consensus protocol implemented will time out requests after a certain amount of time. This timeout defaults to 30 seconds.
- `bdr.raft_log_min_apply_duration` - To move the state machine forward, Raft appends entries to its internal log. During normal operation, appending takes only a few milliseconds. This poses an upper threshold on the duration of that append action, above which an INFO message is logged. This may indicate an actual problem. Default value of this parameter is 3000 ms.
- `bdr.raft_log_min_message_duration` - When to log a consensus request. Measure round trip time of a bdr consensus request and log an INFO message if the time exceeds this parameter. Default value of this parameter is 5000 ms.
- `bdr.raft_group_max_connections` - The maximum number of connections across all BDR groups for a Postgres server. These connections carry bdr consensus requests between the groups' nodes. Default value of this parameter is 100 connections. May only be set at Postgres server start.
- `bdr.backwards_compatibility` - Specifies the version to be backwards-compatible to, in the same numerical format as used by `bdr.bdr_version_num`, e.g. 30618. Enables exact behavior of a former BDR version, even if this has generally unwanted effects. Defaults to the current BDR version. Since this changes from release to release, we advise against explicit use within the configuration file unless the value is different to the current version.
- `bdr.track_replication_estimates` - Track replication estimates in terms of apply rates and catchup intervals for peer nodes. This information can be used by protocols like CAMO to estimate the readiness of a peer node. This parameter is enabled by default. (EDB Postgres Extended)
- `bdr.lag_tracker_apply_rate_weight` - We monitor how far behind peer nodes are in terms of applying WAL from the local node, and calculate a moving average of the apply rates for the lag tracking. This parameter specifies how much contribution newer calculated values have in this moving average calculation. Default value is 0.1. (EDB Postgres Extended)

NODE MANAGEMENT

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

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

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

4.1 Creating and Joining a BDR Group

For BDR, every node has to have a connection to every other node. To make configuration easy, when a new node joins, it automatically configures all existing nodes to connect to it. For this reason, every node, including the first BDR node created, must know the [PostgreSQL connection string](#) (sometimes referred to as a DSN, for “data source name”) that other nodes can use to connect to it. Both formats of connection string are supported. So you can use either key-value format, like `host=myhost port=5432 dbname=mydb`, or URI format: `postgresql://myhost:5432/mydb`.

The SQL function `bdr.create_node_group()` is used to create the BDR group from the local node. Doing so activates BDR on that node and allows other nodes to join the BDR group (which consists of only one node at that point). At the time of creation, you must specify the connection string that other nodes will use to connect to this node.

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

Alternatively, use the command line utility `bdr_init_physical` to create a new node, using `pg_basebackup` (or a physical standby) of an existing node. If using `pg_basebackup`, the `bdr_init_physical` utility can optionally specify the base backup of the target database only, as opposed to the earlier behaviour of backup of the entire database cluster. This should make this activity complete faster, and also allow it to use less space due to the exclusion of unwanted databases. If only the target database is specified, then the excluded databases get cleaned up and removed on the new node.

The `bdr_init_physical` utility replaces the functionality of the `bdr_init_copy` utility from BDR1 and BDR2. It is the BDR3 equivalent of the `pglogical.pglogical_create_subscriber` utility.

!!! Warning Only one node at the time should join the BDR node group, or be parted from it. If a new node is being joined while there is another join or part operation in progress, the new node will sometimes not have consistent data after the join has finished.

When a new BDR node is joined to an existing BDR group or a node is subscribed to an upstream peer, before replication can begin, the system must copy the existing data from the peer node(s) to the local node. This copy must be carefully coordinated so that the local and remote data starts out *identical*; it is not sufficient to just use `pg_dump` yourself. The BDR extension provides built-in facilities for making this initial copy.

During the join process, the BDR extension will synchronize existing data using the provided source node as the basis, and creates all metadata information needed for establishing itself in the mesh topology in the BDR group. If the

connection between the source and the new node disconnects during this initial copy, the join process will need to be restarted from the beginning.

The node that is joining the cluster must not contain any schema or data that already exists on databases in the BDR group. We recommend that the newly joining database is empty except for the BDR and pglogical extension. Ensure that all required database users and roles are created. However, the schema synchronization can be optionally skipped using `synchronize_structure` parameter of `bdr.join_node_group()` function in which case the schema must exist on the newly joining node already.

We recommend that the source node which has the best connection (i.e. is closest) is selected as the source node for joining, since that lowers the time needed for the join to finish.

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

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

Node join can execute concurrently with other node joins for the majority of the time taken to join. Only one regular node at a time can be in either of the states `PROMOTE` or `PROMOTING`, which are typically fairly short. The subscriber-only nodes are an exception to this rule, and they can be concurrently in `PROMOTE` and `PROMOTING` states as well.

Note that the join process uses only one node as the source, so can be executed when nodes are down, if a majority of nodes are available. This can cause a complexity when running logical join: During logical join, the commit timestamp of rows copied from the source node will be set to the latest commit timestamp on the source node. Committed changes on nodes that have a commit timestamp earlier than this (because nodes are down or have significant lag) could conflict with changes from other nodes; in this case, the newly joined node could be resolved differently to other nodes, causing a divergence. As a result, we recommend not to run a node join when significant replication lag exists between nodes; but if this is necessary, run `LiveCompare` on the newly joined node to correct any data divergence once all nodes are available and caught up.

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

4.1.1 Joining a Heterogeneous Cluster

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

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

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

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

4.1.2 Connection DSNs and SSL (TLS)

The DSN of a node is simply a libpq connection string, since nodes connect using libpq. As such, it can contain any permitted libpq connection parameter, including those for SSL. Note that the DSN must work as the connection string from the client connecting to the node in which it is specified. An example of such a set of parameters using a client certificate is shown here:

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

With this setup, the files `bdr_client.crt`, `bdr_client.key` and `root.crt` must be present in the data directory on each node, with the appropriate permissions. For `verify-full` mode, the server's SSL certificate will be checked to ensure that it is directly or indirectly signed with the `root.crt` Certificate Authority, and that the host name or address used in the connection matches the contents of the certificate. In the case of a name, this can match a Subject Alternative Name or, if there are no such names in the certificate, the Subject's Common Name (CN) field. Postgres does not currently support Subject Alternative Names for IP addresses, so if the connection is made by address rather than name, it must match the CN field.

The CN of the client certificate must be the name of the user making the BDR connection. This is usually the user `postgres`. Each node will require matching lines permitting the connection in the `pg_hba.conf` file; for example:

```
hostssl all          postgres 10.1.2.3/24 cert
hostssl replication postgres 10.1.2.3/24 cert
```

Another setup could be to use SCRAM-SHA-256 passwords instead of client certificates, and not bother about verifying the server identity as long as the certificate is properly signed. Here the DSN parameters might be just:

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

...and the corresponding `pg_hba.conf` lines would be like this:

```
hostssl all          postgres 10.1.2.3/24 scram-sha-256
hostssl replication postgres 10.1.2.3/24 scram-sha-256
```

In such a scenario, the `postgres` user would need a `.pgpass` file containing the correct password.

4.2 Witness Nodes

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

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

4.3 Logical Standby Nodes

BDR allows you to create a “logical standby node”, also known as an “offload node”, a “read-only node”, “receive-only node” or “logical read replicas”. A master node can have zero, one or more logical standby nodes.

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

Later, if desired, use `bdr.promote_node()` to move the logical standby into a full, normal send/receive node.

A logical standby is sent data by one source node, defined by the DSN in `bdr.join_node_group`. Changes from all other nodes are received from this one source node, minimizing bandwidth between multiple sites.

There are multiple options for high availability:

- If the source node dies, one physical standby can be promoted to a master. In this case, the new master can continue to feed any/all logical standby nodes.
- If the source node dies, one logical standby can be promoted to a full node and replace the source in a failover operation similar to single master operation. Note that if there are multiple logical standby nodes, the other nodes cannot follow the new master, so the effectiveness of this technique is effectively limited to just one logical standby.

Note that in case a new standby is created of an existing BDR node, the necessary replication slots for operation are not synced to the new standby until at least 16 MB of LSN has elapsed since the group slot was last advanced. In extreme cases, this may require a full 16 MB before slots are synced/created on the streaming replica. If a failover or switchover occurs during this interval, the streaming standby cannot be promoted to replace its BDR node, as the group slot and other dependent slots do not exist yet.

On EDB Postgres Extended and EDB Postgres Advanced, this is resolved automatically. The slot sync-up process on the standby solves this by invoking a function on the upstream. This function moves the group slot in the entire BDR cluster by performing WAL switches and requesting all BDR peer nodes to replay their progress updates. The above causes the group slot to move ahead in a short timespan. This reduces the time required by the standby for the initial slot's sync-up, allowing for faster failover to it, if required.

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

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

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

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

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

Logical standby nodes can themselves be protected using physical standby nodes, if desired, so Master->LogicalStandby->PhysicalStandby. Note that you cannot cascade from LogicalStandby to LogicalStandby.

Note that a logical standby does allow write transactions, so the restrictions of a physical standby do not apply. This can be used to great benefit, since it allows the logical standby to have additional indexes, longer retention periods for

data, intermediate work tables, LISTEN/NOTIFY, temp tables, materialized views, and other differences.

Any changes made locally to logical standbys that commit before the promotion will not be sent to other nodes. All transactions that commit after promotion will be sent onwards. If you perform writes to a logical standby, you are advised to take care to quiesce the database before promotion.

You may make DDL changes to logical standby nodes but they will not be replicated, nor will they attempt to take global DDL locks. BDR functions which act similarly to DDL will also not be replicated. See [DDL Replication]. If you have made incompatible DDL changes to a logical standby, then the database is said to be a divergent node. Promotion of a divergent node will currently result in replication failing. As a result, you should plan to either ensure that a logical standby node is kept free of divergent changes if you intend to use it as a standby, or ensure that divergent nodes are never promoted.

4.4 Physical Standby Nodes

BDR also enables the creation of traditional physical standby failover nodes. These are commonly intended to directly replace a BDR node within the cluster after a short promotion procedure. As with any standard Postgres cluster, a node may have any number of these physical replicas.

There are, however, some minimal prerequisites for this to work properly due to the use of replication slots and other functional requirements in BDR:

- The connection between BDR Primary and Standby uses streaming replication through a physical replication slot.
- The Standby has:
 - `recovery.conf` (for PostgreSQL <12, for PostgreSQL 12+ these settings should be in `postgres.conf`):
 - * `primary_conninfo` pointing to the Primary
 - * `primary_slot_name` naming a physical replication slot on the Primary to be used only by this Standby
 - `postgresql.conf`:
 - * `shared_preload_libraries = 'pglogical, bdr'` at minimum
 - * `hot_standby = on`
 - * `hot_standby_feedback = on`
- The Primary has:
 - `postgresql.conf`:
 - * `pglogical.standby_slot_names` should specify the physical replication slot used for the Standby's `primary_slot_name`.

While this is enough to produce a working physical standby of a BDR node, there are some additional concerns that should be addressed.

Once established, the Standby requires sufficient time and WAL traffic to trigger an initial copy of the Primary's other BDR-related replication slots, including the BDR group slot. At minimum, slots on a Standby are only "live" and will survive a failover if they report a non-zero `confirmed_flush_lsn` as reported by `pg_replication_slots`.

As a consequence, physical standby nodes in newly initialized BDR clusters with low amounts of write activity should be checked before assuming a failover will work normally. Failing to take this precaution can result in the Standby having an incomplete subset of required replication slots necessary to function as a BDR node, and thus an aborted failover.

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

For these reasons it's generally recommended to use either logical standby nodes or subscribe-only group instead of physical standby nodes because they both have better operational characteristics in comparison.

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

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

Upon failover, the Standby must perform one of two actions to replace the Primary:

1. Assume control of the same IP address or hostname as the Primary.
2. Inform the BDR cluster of the change in address by executing the `bdr.alter_node_interface` function on all other BDR nodes.

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

Take special note of the `pglogical.standby_slot_names` parameter as well. While this is a `pglogical` configuration parameter, it is important to set in a BDR cluster where there is a Primary -> Physical Standby relationship. While `pglogical` uses this to ensure physical standby servers always receive WAL traffic before logical replicas, the BDR use case is different.

BDR maintains a group slot that always reflects the state of the cluster node showing the most lag for any outbound replication. With the addition of a physical replica, BDR must be informed that there is a non-participating node member that will, regardless, affect the state of the group slot.

Since the Standby does not directly communicate with the other BDR nodes, the `standby_slot_names` parameter informs BDR to consider named slots as necessary constraints on the group slot as well. When set, the group slot will be held if the Standby shows lag, even if the group slot would have normally been advanced.

As with any physical replica, this type of standby may also be configured as a synchronous replica. As a reminder, this requires:

- On the Standby:
 - Specifying a unique `application_name` in `primary_conninfo`
- On the Primary:
 - Enabling `synchronous_commit`
 - Including the Standby `application_name` in `synchronous_standby_names`

It is possible to mix physical Standby and other BDR nodes in `synchronous_standby_names`. CAMO and Eager All Node Replication use different synchronization mechanisms and do not work with synchronous replication. Please make sure `synchronous_standby_names` does not include the CAMO partner (if CAMO is used) or no BDR node at all (if Eager All Node Replication is used), but only non-BDR nodes, e.g. a the Physical Standby.

4.5 Sub-Groups

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

4.5.1 Subscriber-Only Groups

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

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

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

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

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

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

Notes

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

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

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

4.6 Decoding Worker

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

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

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

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

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

- `pglogical.enable_wal_decoder` - when turned `false`, all WAL Senders using LCRs are restarted to use WAL directly. When `true` along with the BDR group config, a Decoding Worker process is started to produce LCR and WAL Senders use LCR.
- `bdr.receive_lcr` - when `true` on the subscribing node, it requests WAL Sender on the publisher node to use LCRs if available.

4.6.1 Notes

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

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

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

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

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

4.7 Node Restart and Down Node Recovery

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

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

If the amount of write activity on each node is not uniform, the catchup period from nodes with more data could take significantly longer than other nodes. Eventually, the slot state will change to `bdr.node_slots.state = streaming`.

Nodes that are offline for longer periods of time, such as hours or days, can begin to cause resource issues for various reasons. Users should not plan on extended outages without understanding the following issues.

Each node retains change information (using one [replication slot](#) for each peer node) so it can later replay changes to a temporarily unreachable node. If a peer node remains offline indefinitely, this accumulated change information will eventually cause the node to run out of storage space for PostgreSQL transaction logs (WAL in `pg_wal`), and will likely cause the database server to shut down with an error similar to this:

```
PANIC: could not write to file "pg_wal/xlogtemp.559": No space left on device
```

...or report other out-of-disk related symptoms.

In addition, slots for offline nodes also hold back the catalog xmin, preventing vacuuming of catalog tables.

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

Administrators should monitor for node outages (see: [monitoring](#)) and make sure nodes have sufficient free disk space. If the workload is predictable, it may be possible to calculate how much space is used over time, allowing a prediction of the maximum time a node can be down before critical issues arise.

Replication slots created by BDR must not be removed manually. Should that happen, the cluster is damaged and the node that was using the slot must be parted from the cluster, as described below.

Note that while a node is offline, the other nodes may not yet have received the same set of data from the offline node, so this may appear as a slight divergence across nodes. This imbalance across nodes is corrected automatically during the parting process. Later versions may do this at an earlier time.

4.7.1 Replication Slots created by BDR

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

- One *group slot*, named `bdr_<database name>_<group name>`;
- N-1 *node slots*, named `bdr_<database name>_<group name>_<node name>`, where N is the total number of BDR nodes in the cluster, including direct logical standbys, if any.

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

On the other hand, replication slots required by software like Barman or pglogical can be created or dropped, using the appropriate commands for the software, without any effect on BDR. Ensure that slot names used by other software do **not** begin with the prefix `bdr_`.

For example, in a cluster composed by 3 nodes `alpha`, `beta` and `gamma`, where BDR is used to replicate the `mydb` database, and the BDR group is called `mygroup`:

- Node `alpha` has three slots:

- One group slot named `bdr_mydb_mygroup`
- Two node slots named `bdr_mydb_mygroup_beta` and `bdr_mydb_mygroup_gamma`
- Node `beta` has three slots:
 - One group slot named `bdr_mydb_mygroup`
 - Two node slots named `bdr_mydb_mygroup_alpha` and `bdr_mydb_mygroup_gamma`
- Node `gamma` has three slots:
 - One group slot named `bdr_mydb_mygroup`
 - Two node slots named `bdr_mydb_mygroup_alpha` and `bdr_mydb_mygroup_beta`

Group Replication Slot

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

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

The group slot can:

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

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

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

4.7.2 Hashing Long Identifiers

Note that the name of a replication slot - like any other PostgreSQL identifier - cannot be longer than 63 bytes; BDR handles this by shortening the database name, the BDR group name and the name of the node, in case the resulting slot name is too long for that limit. The shortening of an identifier is carried out by replacing the final section of the string with a hash of the string itself.

As an example of this, consider a cluster that replicates a database named `db20xxxxxxxxxxxxxxxxxx` (20 bytes long) using a BDR group named `group20xxxxxxxxxxxxxxxxxx` (20 bytes long); the logical replication slot associated to node `a30xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx` (30 bytes long) will be called:

`bdr_db20xxxx3597186_group20xbe9cbd0_a30xxxxxxxxxxxxx7f304a2`

...since `3597186`, `be9cbd0` and `7f304a2` are respectively the hashes of `db20xxxxxxxxxxxxxxxxxx`, `group20xxxxxxxxxxxxxxxxxx` and `a30xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx`.

4.8 Removing a Node From a BDR Group

Since BDR is designed to recover from extended node outages, you must explicitly tell the system if you are removing a node permanently. If you permanently shut down a node and do not tell the other nodes, then performance will suffer, and eventually the whole system will stop working.

Node removal, also called *parting*, is done using the `bdr.part_node()` function. You must specify the node name (as passed during node creation) to remove a node. The `bdr.part_node()` function can be called from any active node in the BDR group, including the node that is being removed.

Just like the join procedure, parting is done using Raft consensus and requires a majority of nodes to be online to work.

The parting process affects all nodes. The Raft leader will manage a vote between nodes to see which node has the most recent data from the parting node. Then all remaining nodes will make a secondary, temporary, connection to the most-recent node to allow them to catch up any missing data.

A parted node still is known to BDR, but won't consume resources. A node may well be re-added under the very same name as a parted node. In rare cases, it may be advisable to clear all metadata of a parted node with the function `bdr.drop_node()`.

4.8.1 Uninstalling BDR

Dropping the BDR extension will remove all the BDR objects in a node, including metadata tables. This can be done with the following command:

```
DROP EXTENSION bdr;
```

If the database depends on some BDR-specific objects, then the BDR extension cannot be dropped. Examples include:

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

Those dependencies must be removed before dropping the BDR extension, for instance by dropping the dependent objects, altering the column type to a non-BDR equivalent, or changing the sequence type back to `local`.

!!! Warning Dropping the BDR extension **must only** be performed if the node has been successfully parted from its BDR node group, or if it is the last node in the group: dropping BDR and pglogical metadata will break replication to/from the other nodes.

!!! Warning When dropping a local BDR node, or the BDR extension in the local database, any preexisting session might still try to execute a BDR specific workflow, and therefore fail. The problem can be solved by disconnecting the session and then reconnecting the client, or by restarting the instance.

Moreover, the “could not open relation with OID (...)” error could occur when (1) parting a node from a BDR cluster, then (2) dropping the BDR extension (3) recreating it, and finally (4) running `pglogical.replication_set_add_all_tables()`. Restarting the instance will solve the problem.

Similar considerations apply to the pglogical extension, which is required by BDR.

If pglogical is only used by BDR, then it is possible to drop both extensions with a single statement:

```
DROP EXTENSION pglogical, bdr;
```

Conversely, if the node is also using pglogical independently of BDR, e.g. for one-way replication of some tables to a remote database, then only the BDR extension should be dropped.

!!! Warning Dropping BDR from a database that independently uses pglogical can block an existing pglogical subscription from working further with the “BDR global lock manager not initialized yet” error. Restarting the instance will solve the problem.

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

4.9 Listing BDR Topology

4.9.1 Listing BDR Groups

The following (simple) query lists all the BDR node groups of which the current node is a member (will currently return one row only):

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

The configuration of each node group can be displayed using a more complex query:

```
SELECT g.node_group_name
, ns.pub_repsets
, ns.sub_repsets
, g.node_group_default_repset AS default_repset
, node_group_check_constraints AS check_constraints
FROM bdr.local_node_summary ns
JOIN bdr.node_group g USING (node_group_name);
```

4.9.2 Listing Nodes in a BDR Group

The list of all nodes in a given node group (e.g. `mygroup`) can be extracted from the `bdr.node_summary` view as shown in the following example:

```
SELECT node_name      AS name
, node_seq_id        AS ord
, peer_state_name     AS current_state
, peer_target_state_name AS target_state
, interface_connstr   AS dsn
FROM bdr.node_summary
WHERE node_group_name = 'mygroup';
```

Note that the read-only state of a node, as shown in the `current_state` or in the `target_state` query columns, is indicated as `STANDBY`.

4.9.3 List of Node States

- **NONE:** Node state is unset when the worker starts, expected to be set quickly to the current known state.
- **CREATED:** `bdr.create_node()` has been executed, but the node is not a member of any BDR cluster yet.
- **JOIN_START:** `bdr.join_node_group()` begins to join the local node to an existing BDR cluster.
- **JOINING:** The node join has started and is currently at the initial sync phase, creating the schema and data on the node.
- **CATCHUP:** Initial sync phase is completed; now the join is at the last step of retrieving and applying transactions that were performed on the upstream peer node since the join started.
- **STANDBY:** Node join has finished, but not yet started to broadcast changes. All joins spend some time in this state, but if defined as a Logical Standby, the node will continue in this state.
- **PROMOTE:** Node was a logical standby and we just called `bdr.promote_node` to move the node state to **ACTIVE**. These two **PROMOTE** states have to be coherent to the fact, that only one node can be with a state higher than **STANDBY** but lower than **ACTIVE**.
- **PROMOTING:** Promotion from logical standby to full BDR node is in progress.
- **ACTIVE:** The node is a full BDR node and is currently **ACTIVE**. This is the most common node status.
- **PART_START:** Node was **ACTIVE** or **STANDBY** and we just called `bdr.part_node` to remove the node from the BDR cluster.
- **PARTING:** Node disconnects from other nodes and plays no further part in consensus or replication.
- **PART_CATCHUP:** Non-parting nodes synchronize any missing data from the recently parted node.
- **PARTED:** Node parting operation is now complete on all nodes.

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

4.10 Managing Shard Groups

BDR clusters may contain an array of Shard Groups for the AutoScale feature. These are shown as a sub-node group that is composed of an array of sub-sub node groups known as Shard Groups.

Operations that can be performed on the Shard Group are:

- Create Shard Array
- Drop Shard Array
- Repair - add new nodes to replace failed nodes
- Expand - add new Shard Groups
- Re-Balance - re-distribute data across Shard Groups

4.10.1 Create/Drop

4.10.2 Expand

e.g. expand from 4 Shard Groups to 8 Shard Groups

This operation can occur without interfering with user operations.

4.10.3 Re-Balance

e.g. move data from where it was in a 4-node array to how it would be ideally placed in an 8-node array.

Some portion of the data is moved from one Shard Group to another, so this action can take an extended period, depending upon how much data is to be moved. The data is moved one partition at a time, so is restartable without too much wasted effort.

Note that re-balancing is optional.

This operation can occur without interfering with user operations, even when this includes write transactions.

4.11 Node Management Interfaces

Nodes can be added and removed dynamically using the SQL interfaces.

4.11.1 bdr.create_node

This function creates a node.

Synopsis

```
bdr.create_node(node_name text, local_dsn text)
```

Parameters

- **node_name** - name of the new node; only one node is allowed per database. Valid node names consist of lower case letters, numbers, hyphens and underscores.
- **local_dsn** - connection string to the node

Notes

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

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

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

4.11.2 bdr.drop_node

Drops a node. This function is *not intended for regular use* and should only be executed under the instructions of Technical Support.

This function removes the metadata for a given node from the local database. The node can be either:

- The **local** node, in which case all the node metadata is removed, including information about remote nodes;
- A **remote** node, in which case only metadata for that specific node is removed.

Synopsis

```
bdr.drop_node(node_name text, cascade boolean DEFAULT false, force boolean DEFAULT false)
```

Parameters

- **node_name** - Name of an existing node.
- **cascade** - Whether to cascade to dependent objects, this will also delete the associated pglogical node. This option should be used with caution!
- **force** - Circumvents all sanity checks and forces the removal of all metadata for the given BDR node despite a possible danger of causing inconsistencies. A forced node drop is to be used by Technical Support only in case of emergencies related to parting.

Notes

Before you run this, you should already have parted the node using `bdr.part_node()`.

This function removes metadata for a given node from the local database. The node can be either the local node, in which case all the node metadata are removed, including info about remote nodes are removed; or it can be the remote node, in which case only metadata for that specific node is removed.

!!! Note BDR3 can have a maximum of 1024 node records (both ACTIVE and PARTED) at one time. This is because each node has a unique sequence number assigned to it, for use by timeshard sequences. PARTED nodes are not automatically cleaned up at the moment; should this become a problem, this function can be used to remove those records.

4.11.3 bdr.create_node_group

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

Synopsis

```
bdr.create_node_group(node_group_name text,
                     parent_group_name text,
                     join_node_group boolean DEFAULT true,
                     node_group_type text DEFAULT NULL)
```

Parameters

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

Notes

This function will pass request to local consensus worker that is running for the local node.

The function is not transactional. The creation of the group is a background process, so once the function has finished, the changes cannot be rolled back. Also, the changes might not be immediately visible to the current transaction; the `bdr.wait_for_join_completion` can be called to wait until they are.

The group creation does not hold any locks.

4.11.4 `bdr.alter_node_group_config`

This function changes the configuration parameter(s) of an existing BDR group. Options with NULL value (default for all of them) will not be modified.

Synopsis

```
bdr.alter_node_group_config(node_group_name text,  
                             insert_to_update boolean DEFAULT NULL,  
                             update_to_insert boolean DEFAULT NULL,  
                             ignore_redundant_updates boolean DEFAULT NULL,  
                             check_full_tuple boolean DEFAULT NULL,  
                             apply_delay interval DEFAULT NULL,  
                             check_constraints boolean DEFAULT NULL,  
                             num_writers int DEFAULT NULL,  
                             enable_wal_decoder boolean DEFAULT NULL)
```

Parameters

- `node_group_name` - Name of an existing BDR group; the local node must be part of the group.
- `insert_to_update` - Reserved for backwards compatibility reasons.
- `update_to_insert` - Reserved for backwards compatibility reasons. versions of BDR. Use `bdr.alter_node_set_conflict_resolver` instead.**
- `ignore_redundant_updates` - Reserved for backwards compatibility reasons.
- `check_full_tuple` - Reserved for backwards compatibility reasons.
- `apply_delay` - Reserved for backwards compatibility reasons
- `check_constraints` - Whether the apply process will check the constraints when writing replicated data. **This option is deprecated and may be disabled or removed in future versions of BDR.**
- `num_writers` - number of parallel writers for subscription backing this node group, -1 means the default (as specified by the pglogical GUC `pglogical.writers_per_subscription`) will be used. Valid values are either -1 or a positive integer.
- `enable_wal_decoder` - Enables/disables the Decoding Worker process.

Note that all of the options parameters are simply used to control the pglogical writer.

Notes

This function will pass a request to the group consensus mechanism to change the defaults. The changes made are replicated globally via the consensus mechanism.

The function is not transactional. The request is processed in the background so the function call cannot be rolled back. Also, the changes may not be immediately visible to the current transaction.

This function does not hold any locks.

!!! Warning When this function is used to change the `apply_delay` value, the change does not apply to nodes that are already members of the group.

Note that this restriction has little consequence on production usage, because this value is normally not used outside of testing.

4.11.5 bdr.join_node_group

This function joins the local node to an already existing BDR group.

Synopsis

```
bdr.join_node_group (
    join_target_dsn text,
    node_group_name text DEFAULT NULL,
    pause_in_standby boolean DEFAULT false,
    wait_for_completion boolean DEFAULT true,
    synchronize_structure text DEFAULT 'all'
)
```

Parameters

- `join_target_dsn` - Specifies the connection string to existing (source) node in the BDR group we wish to add the local node to.
- `node_group_name` - Optional name of the BDR group; defaults to NULL which tries to autodetect the group name from information present on the source node.
- `pause_in_standby` - Optionally tells the join process to only join as a logical standby node, which can be later promoted to a full member.
- `wait_for_completion` - Wait for the join process to complete before returning; defaults to true.
- `synchronize_structure` - Set what kind of structure (schema) synchronization should be done during the join. Valid options are 'all' which synchronizes the complete database structure, and 'none' which will not synchronize any structure (however, it will still synchronize data).

If `wait_for_completion` is specified as false; this is an asynchronous call which returns as soon as the joining procedure has started. Progress of the join can be seen in logs and the `bdr.state_journal_details` information view, or by calling the `bdr.wait_for_join_completion()` function once `bdr.join_node_group()` returns.

Notes

This function will pass a request to the group consensus mechanism via the node that the `join_target_dsn` connection string points to. The changes made are replicated globally via the consensus mechanism.

The function is not transactional. The joining process happens in the background and as such cannot be rolled back. The changes are only visible to the local transaction if `wait_for_completion` was set to true or by calling `bdr.wait_for_join_completion` later.

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

Node join does not hold any locks in the BDR group.

4.11.6 bdr.promote_node

This function promotes a local logical standby node to a full member of the BDR group.

Synopsis

```
bdr.promote_node(wait_for_completion boolean DEFAULT true)
```

Notes

This function will pass a request to the group consensus mechanism to change the defaults. The changes made are replicated globally via the consensus mechanism.

The function is not transactional. The promotion process happens in the background, and as such cannot be rolled back. The changes are only visible to the local transaction if `wait_for_completion` was set to true or by calling `bdr.wait_for_join_completion` later.

The promotion process holds lock against other promotions. This lock will not block other `bdr.promote_node` calls, but will prevent the background process of promotion from moving forward on more than one node at a time.

4.11.7 bdr.wait_for_join_completion

This function waits for the join procedure of a local node to finish.

Synopsis

```
bdr.wait_for_join_completion(verbose_progress boolean DEFAULT false)
```

Parameters

- `verbose_progress` - Optionally prints information about individual steps taken during the join procedure.

Notes

This function waits until the checks state of the local node reaches the target state, which was set by `bdr.create_node_group`, `bdr.join_node_group` or `bdr.promote_node`.

4.11.8 bdr.part_node

Removes (“parts”) the node from the BDR group, but does not remove data from the node.

The function can be called from any active node in the BDR group, including the node which is being removed. However, just to make it clear, once the node is *PARTED* it can not *part* other nodes in the cluster.

!!! Note you are *parting* the local node you must set `wait_for_completion` false, otherwise it will error.

!!! Warning `s` action is permanent. If you wish to temporarily halt replication a node, see `bdr.alter_subscription_disable()`.

Synopsis

```
bdr.part_node (
    node_name text,
    wait_for_completion boolean DEFAULT true,
    force boolean DEFAULT false
)
```

Parameters

- `node_name` - Name of an existing node to part.
- `wait_for_completion` - If true, the function will not return until the node is fully parted from the cluster, otherwise the function will just start the parting procedure and return immediately without waiting. Always set to false when executing on the local node, or when using force.
- `force` - Forces removal of the node on the local node. This will set the node state locally if consensus could not be reached or if the node parting process has stuck.

!!! Warning Using `force = true` may leave the BDR group in a inconsistent state and should be only used to recover from byzantine failures where it's impossible to remove the node any other way.**

Notes

This function will pass a request to the group consensus mechanism to part the given node. The changes made are replicated globally via the consensus mechanism. The parting process happens in the background, and as such cannot be rolled back. The changes made by the parting process are only visible to the local transaction if `wait_for_completion` was set to `true`.

With `force` set to `true`, on consensus failure, this function will set the state of the given node only on the local node. In such a case, the function is transactional (because the function itself changes the node state) and can be rolled back. If the function is called on a node which is already in process of parting with `force` set to `true`, it will also just mark the given node as parted locally and exit. This is only useful when the consensus cannot be reached on the cluster (i.e. the majority of the nodes are down) or if the parting process gets stuck for whatever reason. But it is important to take into account that when the parting node that was receiving writes, the parting process may take a long time without being stuck, as the other nodes need to resynchronize any missing data from the given node. The force parting completely skips this resynchronization, and as such can leave the other nodes in inconsistent state.

The parting process does not hold any locks.

4.11.9 `bdr.alter_node_interface`

This function changes the connection string (DSN) of a specified node.

Synopsis

```
bdr.alter_node_interface(node_name text, interface_dsn text)
```

Parameters

- `node_name` - name of an existing node to alter
- `interface_dsn` - new connection string for a node

Notes

This function is only run on the local node and the changes are only made on the local node. This means that it should normally be executed on every node in the BDR group, including the node which is being changed.

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

The function holds lock on the local node.

4.11.10 `bdr.alter_subscription_enable`

This function enables either the specified subscription or all the subscriptions of the local BDR node. Also known as resume subscription. No error is thrown if the subscription is already enabled. Returns the number of subscriptions affected by this operation.

Synopsis

```
bdr.alter_subscription_enable(
    subscription_name name DEFAULT NULL,
    immediate boolean DEFAULT false
)
```

Parameters

- `subscription_name` - Name of the subscription to enable; if NULL (the default), all subscriptions on the local node will be enabled.
- `immediate` - This currently has no effect.

Notes

This function is not replicated and only affects local node subscriptions (either a specific node or all nodes).

This function is transactional - it can be rolled back and any catalog changes can be seen by the current transaction. The subscription workers will be started by a background process after the transaction has committed.

4.11.11 bdr.alter_subscription_disable

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

Synopsis

```
bdr.alter_subscription_disable(
    subscription_name name DEFAULT NULL,
    immediate boolean DEFAULT false
)
```

Parameters

- `subscription_name` - Name of the subscription to disable; if NULL (the default), all subscriptions on the local node will be disabled.
- `immediate` - Immediate is used to force the action immediately, stopping all the workers associated with the disabled subscription. With this option true, this function cannot be run inside of the transaction block.

Notes

This function is not replicated and only affects local node subscriptions (either a specific subscription or all subscriptions).

This function is transactional - it can be rolled back and any catalog changes can be seen by the current transaction. However, the timing of the subscription worker stopping depends on the value of `immediate`; if set to `true`, the workers will be stopped immediately; if set to `false`, they will be stopped at the `COMMIT` time.

4.12 Node Management Commands

BDR also provides a command line utility for adding nodes to the BDR group via physical copy (`pg_basebackup`) of an existing node, and for converting a physical standby of an existing node to a new node in the BDR group.

4.12.1 `bdr_init_physical`

This is a regular command which is added to PostgreSQL's bin directory.

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

When starting from an empty data directory, if the selective backup option is chosen, then only that database will be copied from the source node. The excluded databases will be dropped and cleaned up on the new node (EDB Postgres Extended).

If the specified data directory is non-empty, this will be used as the base for the new node. If the data directory is already active as a physical standby node, it is required to stop the standby before running `bdr_init_physical`, which will manage Postgres itself. Initially it will wait for catchup and then promote to a master node before joining the BDR group. Note that the `--standby` option, if used, will turn the existing physical standby into a logical standby node; it refers to the end state of the new BDR node, not the starting state of the specified data directory.

This command will drop all `pglogical`-only subscriptions and configuration from the database and will also drop all PostgreSQL native logical replication subscriptions from the database (or just disable them when the `-S` option is used), as well as any replication origins and slots.

It is the BDR3 version of the `pglogical_create_subscriber` utility.

Synopsis

`bdr_init_physical [OPTION] ...`

Options

General Options

- `-D, --pgdata=DIRECTORY` - The data directory to be used for the new node; it can be either empty/non-existing directory, or a directory populated using the `pg_basebackup -X stream` command (required).
- `-l, --log-file=FILE` - Use `FILE` for logging; default is `bdr_init_physical_postgres.log`.
- `-n, --node-name=NAME` - The name of the newly created node (required).

- `--replication-sets=SETS` - The name of a comma-separated list of replication set names to use; all replication sets will be used if not specified.
- `--standby` - Create a logical standby (receive only node) rather than full send/receive node.
- `--node-group-name` - Group to join, defaults to the same group as source node.
- `-s`, `--stop` - Stop the server once the initialization is done.
- `-v` - Increase logging verbosity.
- `-L` - Perform selective `pg_basebackup` when used in conjunction with an empty/non-existing data directory (`-D` option). (EDB Postgres Extended)
- `-S` - Instead of dropping logical replication subscriptions, just disable them.

Connection Options

- `-d`, `--remote-dsn=CONNSTR` - connection string for remote node (required)
- `--local-dsn=CONNSTR` - connection string for local node (required)

Configuration Files Override

- `--hba-conf -path` to the new `pg_hba.conf`
- `--postgresql-conf -path` to the new `postgresql.conf`

Notes

The replication set names specified in the command do not affect the data that exists in the data directory before the node joins the BDR group. This is true whether `bdr_init_physical` makes its own basebackup or an existing base backup is being promoted to a new BDR node. Thus the `--replication-sets` option only affects the data published and subscribed-to after the node joins the BDR node group. This behaviour is different from the way replication sets are used in a logical join i.e. when using `bdr.join_node_group()`.

Unwanted tables may be truncated by the operator after the join has completed. Refer to the `bdr.tables` catalog to determine replication set membership and identify tables that are not members of any subscribed-to replication set. It's strongly recommended that you truncate the tables rather than drop them, because:

1. DDL replication sets are not necessarily the same as row (DML) replication sets, so you could inadvertently drop the table on other nodes;
2. If you later want to add the table to a replication set and you have dropped it on some subset of nodes, you will need to take care to re-create it only on those nodes without creating DDL conflicts before you can add it to any replication sets.

It's much simpler and safer to truncate your non-replicated tables, leaving them present but empty.

A future version of BDR may automatically omit or remove tables that are not part of the selected replication set(s) for a physical join, so your application should not rely on details of the behaviour documented here.

DDL REPLICATION

DDL stands for “Data Definition Language”: the subset of the SQL language that creates, alters and drops database objects.

For operational convenience and correctness, BDR replicates most DDL actions, with these exceptions:

- Temporary or Unlogged relations
- Certain, mostly long-running DDL statements (see list below)
- Locking commands (LOCK)
- Table Maintenance commands (VACUUM, ANALYZE, CLUSTER, REINDEX)
- Actions of autovacuum
- Operational commands (CHECKPOINT, ALTER SYSTEM)
- Actions related to Databases or Tablespaces

Automatic DDL replication makes it easier to make certain DDL changes without having to manually distribute the DDL change to all nodes and ensure that they are consistent.

In the default replication set, DDL is replicated to all nodes by default. To replicate DDL, a DDL replication filter has to be added to the replication set. See [DDL Replication Filtering].

BDR is significantly different to standalone PostgreSQL when it comes to DDL replication, and treating it as the same is the most common operational issue with BDR.

The main difference from table replication is that DDL replication does not replicate the result of the DDL, but the statement itself. This works very well in most cases, though introduces the requirement that the DDL must execute similarly on all nodes. A more subtle point is that the DDL must be immutable with respect to all datatype-specific parameter settings, including any datatypes introduced by extensions (i.e. not built-in). For example, the DDL statement must execute correctly in the default encoding used on each node.

5.1 DDL Replication Options

The `bdr.ddl_replication` parameter specifies replication behavior.

`bdr.ddl_replication = on` is the default and will replicate DDL to the default replication set, which by default means all nodes. Non-default replication sets do not replicate DDL, unless they have a [DDL filter](#) defined for them.

You can also replicate DDL to specific replication sets using the function `bdr.replicate_ddl_command()`. This can be helpful if you want to run DDL commands when a node is down, or if you want to have indexes or partitions that exist on a subset of nodes or rep sets, e.g. all nodes at site1.

```
SELECT bdr.replicate_ddl_command(  
    'CREATE INDEX CONCURRENTLY ON foo (col7);',  
    ARRAY['site1'],      -- the replication sets  
    'on');               -- ddl_locking to apply
```

It is possible, but not recommended, to skip automatic DDL replication and execute it manually on each node using `bdr.ddl_replication` configuration parameters.

```
SET bdr.ddl_replication = off;
```

When set, it will make BDR skip both the global locking and the replication of executed DDL commands, so you must then run the DDL manually on all nodes.

!!! Warning Executing DDL manually on each node without global locking can cause the whole BDR group to stop replicating if conflicting DDL or DML is executed concurrently.

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

5.2 Executing DDL on BDR Systems

A BDR group is not the same as a standalone PostgreSQL server. It is based on asynchronous multi-master replication without central locking and without a transaction co-ordinator. This has important implications when executing DDL.

DDL that executes in parallel will continue to do so with BDR. DDL execution will respect the parameters that affect parallel operation on each node as it executes, so differences in the settings between nodes may be noticeable.

Execution of conflicting DDL needs to be prevented, otherwise DDL replication will end up causing errors and the replication will stop.

BDR offers 3 levels of protection against those problems:

`ddl_locking = 'dml'` is the best option for operations, usable when you execute DDL from only one node at a time. This is not the default, but it is recommended that you use this setting if you can control where DDL is executed from, to ensure that there are no inter-node conflicts. Intra-node conflicts are already handled by PostgreSQL.

`ddl_locking = on` is the strictest option, and is best when DDL might be executed from any node concurrently and you would like to ensure correctness.

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

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

When using the `bdr.replicate_ddl_command`, it is possible to set this parameter directly via the third argument, using the specified `bdr.ddl_locking` setting only for the DDL commands passed to that function.

5.3 DDL Locking Details

There are two kinds of locks used to enforce correctness of replicated DDL with BDR.

The first kind is known as a Global DDL Lock, and is only used when `ddl_locking = on`. A Global DDL Lock prevents any other DDL from executing on the cluster while each DDL statement runs. This ensures full correctness in the general case, but is clearly too strict for many simple cases. BDR acquires a global lock on DDL operations the first time in a transaction where schema changes are made. This effectively serializes the DDL-executing transactions in the cluster. In other words, while DDL is running, no other connection on any node can run another DDL command, **even if it affects different table(s)**.

To acquire a lock on DDL operations, the BDR node executing DDL contacts the other nodes in a BDR group and asks them to grant it the exclusive right to execute DDL. The lock request is sent via regular replication stream and the nodes respond via replication stream as well. So it's important that nodes (or at least a majority of the nodes) should be running without much replication delay. Otherwise it may take a very long time for the node to acquire the DDL lock. Once the majority of nodes agrees, the DDL execution is carried out.

The ordering of DDL locking is decided using the Raft protocol. DDL statements executed on one node will be executed in the same sequence on all other nodes.

In order to ensure that the node running a DDL has seen effects of all prior DDLs run in the cluster, it waits until it has caught up with the node that had run the previous DDL. If the node running the current DDL is lagging behind in replication with respect to the node that ran the previous DDL, then it may take very long to acquire the lock. Hence it's preferable to run DDLs from a single node or the nodes which have nearly caught up with replication changes originating at other nodes.

The second kind is known as a Relation DML Lock. This kind of lock is used when either `ddl_locking = on` or `ddl_locking = dml`, and the DDL statement might cause in-flight DML statements to fail, such as when we add or modify a constraint such as a unique constraint, check constraint or NOT NULL constraint. Relation DML locks affect only one relation at a time. Relation DML locks ensure that no DDL executes while there are changes in the queue that might cause replication to halt with an error.

To acquire the global DML lock on a table, the BDR node executing the DDL contacts **all** other nodes in a BDR group, asking them to lock the table against writes, and we wait while all pending changes to that table are drained. Once all nodes are fully caught up, the originator of the DML lock is free to perform schema changes to the table and replicate them to the other nodes.

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

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

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

Locking behavior is specified by the `bdr.ddl_locking` parameter, as explained in *Executing DDL on BDR systems*:

- `ddl_locking = on` takes Global DDL Lock and, if needed, takes Relation DML Lock.
- `ddl_locking = dml` skips Global DDL Lock and, if needed, takes Relation DML Lock.
- `ddl_locking = off` skips both Global DDL Lock and Relation DML Lock.

Note also that some BDR functions make DDL changes, so for those functions, DDL locking behavior applies. This will be noted in the docs for each function.

Thus, `ddl_locking = dml` is safe only when we can guarantee that no conflicting DDL will be executed from other nodes, because with this setting, the statements which only require the Global DDL Lock will not use the global locking at all.

`ddl_locking = off` is safe only when the user can guarantee that there are no conflicting DDL and no conflicting DML operations on the database objects we execute DDL on. If you turn locking off and then experience difficulties, you may lose in-flight changes to data; any issues caused will need to be resolved by the user application team.

In some cases, concurrently executing DDL can properly be serialized. Should these serialization failures occur, the DDL may be re-executed.

DDL replication is not active on Logical Standby nodes until they are promoted.

Note that some BDR management functions act like DDL, meaning that they will attempt to take global locks and their actions will be replicated, if DDL replication is active. The full list of replicated functions is listed in *BDR Functions that behave like DDL*.

DDL executed on temporary tables never need global locks.

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

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

5.4 Minimizing the Impact of DDL

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

- To minimize the impact of DDL, transactions performing DDL should be short, should not be combined with lots of row changes, and should avoid long running foreign key or other constraint re-checks.
- For ALTER TABLE, please use ADD CONSTRAINT NOT VALID, followed by another transaction with VALIDATE CONSTRAINT, rather than just using ADD CONSTRAINT. Note that VALIDATE CONSTRAINT will wait until replayed on all nodes, which gives a noticeable delay to receive confirmations.
- When indexing, use CONCURRENTLY option whenever possible.

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

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

We recommend using the bdr.run_on_all_nodes() technique above with CREATE INDEX CONCURRENTLY, noting that DDL replication must be disabled for whole session because CREATE INDEX CONCURRENTLY is a multi-transaction command. CREATE INDEX should be avoided on production systems since it prevents writes while it executes. REINDEX is replicated in versions up to BDR3.6, but not in BDR3.7 or later. Using REINDEX should be avoided because of the AccessExclusiveLocks it holds.

Instead, REINDEX CONCURRENTLY should be used (or reindexdb -concurrently), which is available in PG12+ or 2QPG11+.

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

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

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

Multiple DDL statements might benefit from bunching into a single transaction rather than fired as individual statements, so the DDL lock only has to be taken once. This may not be desirable if the table-level locks interfere with normal operations.

If DDL is holding the system up for too long, it is possible and safe to cancel the DDL on the originating node as you would cancel any other statement, e.g. with Control-C in psql or with pg_cancel_backend(). You cannot cancel a DDL lock from any other node.

It is possible to control how long the global lock will take with (optional) global locking timeout settings. The `bdr.global_lock_timeout` will limit how long the wait for acquiring the global lock can take before it is cancelled; `bdr.global_lock_statement_timeout` limits the runtime length of any statement in transaction that holds global locks, and `bdr.global_lock_idle_timeout` sets the maximum allowed idle time (time between statements) for a transaction holding any global locks. All of these timeouts can be disabled by setting their values to zero.

Once the DDL operation has committed on the originating node, it cannot be canceled or aborted. The BDR group must wait for it to apply successfully on other nodes that confirmed the global lock and for them to acknowledge replay. This is why it is important to keep DDL transactions short and fast.

5.5 Handling DDL With Down Nodes

If the node initiating the global DDL lock goes down after it has acquired the global lock (either DDL or DML), the lock stays active. The global locks will not time out, even if timeouts have been set. In case the node comes back up, it will automatically release all the global locks that it holds.

If it stays down for a prolonged period time (or forever), remove the node from BDR group in order to release the global locks. This might be one reason for executing emergency DDL using the SET command as the `bdr_superuser` to update the `bdr.ddl_locking` value.

If one of the other nodes goes down after it has confirmed the global lock, but before the command acquiring it has been executed, the execution of that command requesting the lock will continue as if the node was up.

As mentioned in the previous section, the global DDL lock only requires a majority of the nodes to respond, and so it will work if part of the cluster is down, as long as a majority is running and reachable, while the DML lock cannot be acquired unless the whole cluster is available.

If we have the global DDL or global DML lock and another node goes down, the command will continue normally and the lock will be released.

5.6 Statement Specific DDL Replication Concerns

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

BDR prevents some DDL statements from running when it is active on a database. This protects the consistency of the system by disallowing statements that cannot be replicated correctly, or for which replication is not yet supported. Statements that are supported with some restrictions are covered in [DDL Statements With Restrictions]; while commands that are entirely disallowed in BDR are covered in prohibited DDL statements.

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

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

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

5.7 BDR DDL Command Handling Matrix

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

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

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

continues on next page

Table 1 – continued from previous page

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

continues on next page

Table 1 – continued from previous page

Command	Allowed	Replicated	Lock
CREATE QUEUE TABLE	Y	Y	DDL
CREATE REDACTION POLICY	Y	Y	DDL
CREATE RESOURCE GROUP	Y	N	N
CREATE ROLE	Y	Y	DDL
CREATE ROUTINE	Y	Y	DDL
CREATE RULE	Y	Y	DDL
CREATE SCHEMA	Y	Y	DDL
CREATE SEQUENCE	<i>Details</i>	Y	DDL
CREATE SERVER	Y	Y	DDL
CREATE STATISTICS	Y	Y	DDL
CREATE SUBSCRIPTION	Y	Y	DDL
CREATE SYNONYM	Y	Y	DDL
CREATE TABLE	<i>Details</i>	Y	DDL
CREATE TABLE AS	<i>Details</i>	Y	DDL
CREATE TABLESPACE	Y	N	N
CREATE TEXT SEARCH CONFIGURATION	Y	Y	DDL
CREATE TEXT SEARCH DICTIONARY	Y	Y	DDL
CREATE TEXT SEARCH PARSER	Y	Y	DDL
CREATE TEXT SEARCH TEMPLATE	Y	Y	DDL
CREATE TRANSFORM	Y	Y	DDL
CREATE TRIGGER	Y	Y	DDL
CREATE TYPE	Y	Y	DDL
CREATE TYPE BODY	Y	Y	DDL
CREATE USER MAPPING	Y	Y	DDL
CREATE VIEW	Y	Y	DDL
DEALLOCATE	Y	N	N
DEALLOCATE ALL	Y	N	N
DECLARE CURSOR	Y	N	N
DISCARD	Y	N	N
DISCARD ALL	Y	N	N
DISCARD PLANS	Y	N	N
DISCARD SEQUENCES	Y	N	N
DISCARD TEMP	Y	N	N
DO	Y	N	N
DROP ACCESS METHOD	Y	Y	DDL
DROP AGGREGATE	Y	Y	DDL
DROP CAST	Y	Y	DDL
DROP COLLATION	Y	Y	DDL
DROP CONSTRAINT	Y	Y	DDL
DROP CONVERSION	Y	Y	DDL
DROP DATABASE	Y	N	N
DROP DATABASE LINK	Y	Y	DDL
DROP DIRECTORY	Y	Y	DDL
DROP DOMAIN	Y	Y	DDL
DROP EVENT TRIGGER	Y	Y	DDL
DROP EXTENSION	Y	Y	DDL
DROP FOREIGN DATA WRAPPER	Y	Y	DDL
DROP FOREIGN TABLE	Y	Y	DDL
DROP FUNCTION	Y	Y	DDL

continues on next page

Table 1 – continued from previous page

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

continues on next page

Table 1 – continued from previous page

Command	Allowed	Replicated	Lock
NOTIFY	Y	N	N
PREPARE	Y	N	N
PREPARE TRANSACTION	Y	N	N
REASSIGN OWNED	Y	Y	DDL
REFRESH MATERIALIZED VIEW	Y	N	N
REINDEX	Y	N	N
RELEASE	Y	N	N
RESET	Y	N	N
REVOKE	Y	<i>Details</i>	DDL
REVOKE ROLE	Y	Y	DDL
ROLLBACK	Y	N	N
ROLLBACK PREPARED	Y	N	N
SAVEPOINT	Y	N	N
SECURITY LABEL	Y	<i>De tails</i>	DDL
SELECT INTO	<i>Detai ls</i>	Y	DDL
SET	Y	N	N
SET CONSTRAINTS	Y	N	N
SHOW	Y	N	N
START TRANSACTION	Y	N	N
TRUNCATE TABLE	Y	<i>De tails</i>	<i>Details</i>
UNLISTEN	Y	N	N
VACUUM	Y	N	N

5.7.1 ALTER SEQUENCE

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

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

5.7.2 ALTER TABLE

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

ALTER TABLE Disallowed Commands

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

- ADD COLUMN ... DEFAULT (non-immutable expression) - This is not allowed because it would currently result in different data on different nodes. See [Adding a Column](#) for a suggested workaround.
- ADD CONSTRAINT ... EXCLUDE - Exclusion constraints are not supported for now. Exclusion constraints do not make much sense in an asynchronous system and lead to changes that cannot be replayed.
- ALTER TABLE ... SET WITH[OUT] OIDS - Is not supported for the same reasons as in CREATE TABLE.
- ALTER COLUMN ... SET STORAGE external - Will be rejected if the column is one of the columns of the replica identity for the table.
- RENAME - cannot rename an Autopartitioned table.
- SET SCHEMA - cannot set the schema of an Autopartitioned table.

- **ALTER COLUMN ... TYPE** - Changing a column's type is not supported if the command causes the whole table to be rewritten, which occurs when the change is not binary coercible. Note that binary coercible changes may only be allowed one way. For example, the change from VARCHAR(128) to VARCHAR(256) is binary coercible and therefore allowed, whereas the change VARCHAR(256) to VARCHAR(128) is not binary coercible and therefore normally disallowed. For non-replicated **ALTER COLUMN ... TYPE** it can be allowed if the column is automatically castable to the new type (it does not contain the USING clause). See below for an example. Table rewrites would hold an AccessExclusiveLock for extended periods on larger tables, so such commands are likely to be infeasible on highly available databases in any case. See [Changing a Column's Type](#) for a suggested workarounds.
- **ALTER TABLE ... ADD FOREIGN KEY** - Is not supported if current user does not have permission to read the referenced table, or if the referenced table has RLS restrictions enabled which current user cannot bypass.

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

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

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

ALTER TABLE Locking

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

- **ALTER TABLE ... ADD COLUMN ... (immutable) DEFAULT**
- **ALTER TABLE ... ALTER COLUMN ... SET DEFAULT expression**
- **ALTER TABLE ... ALTER COLUMN ... DROP DEFAULT**
- **ALTER TABLE ... ALTER COLUMN ... TYPE** if it does not require rewrite (currently only available on EDB Postgres Extended and EDB Postgres Advanced)
- **ALTER TABLE ... ALTER COLUMN ... SET STATISTICS**
- **ALTER TABLE ... VALIDATE CONSTRAINT**
- **ALTER TABLE ... ATTACH PARTITION**
- **ALTER TABLE ... DETACH PARTITION**
- **ALTER TABLE ... ENABLE TRIGGER** (ENABLE REPLICA TRIGGER will still take a DML lock)
- **ALTER TABLE ... CLUSTER ON**
- **ALTER TABLE ... SET WITHOUT CLUSTER**
- **ALTER TABLE ... SET (storage_parameter = value [, ...])**
- **ALTER TABLE ... RESET (storage_parameter = [, ...])**
- **ALTER TABLE ... OWNER TO**

All other variants of **ALTER TABLE** take a DML lock on the table being modified. Some variants of **ALTER TABLE** have restrictions, noted below.

ALTER TABLE Examples

This next example works because the type change is binary coercible and so does not cause a table rewrite, so it will execute as a catalog-only change.

```
CREATE TABLE foo (id BIGINT PRIMARY KEY, description VARCHAR(20));
ALTER TABLE foo ALTER COLUMN description TYPE VARCHAR(128);
```

However, making this change to reverse the above command is not possible because the change from VARCHAR(128) to VARCHAR(20) is not binary coercible.

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

See later for suggested workarounds.

It is useful to provide context for different types of ALTER TABLE ... ALTER COLUMN TYPE (ATCT) operations that are possible in general and in non-replicated environments.

Some ATCT operations only update the metadata of the underlying column type and do not require a rewrite of the underlying table data. This is typically the case when the existing column type and the target type are binary coercible. For example:

```
CREATE TABLE sample (col1 BIGINT PRIMARY KEY, col2 VARCHAR(128), col3 INT);
ALTER TABLE sample ALTER COLUMN col2 TYPE VARCHAR(256);
```

It will also be OK to change the column type to VARCHAR or TEXT datatypes because of binary coercibility. Again, this is just a metadata update of the underlying column type.

```
ALTER TABLE sample ALTER COLUMN col2 TYPE VARCHAR;
ALTER TABLE sample ALTER COLUMN col2 TYPE TEXT;
```

However, if you want to reduce the size of col2, then that will lead to a rewrite of the underlying table data. Rewrite of a table is normally restricted.

```
ALTER TABLE sample ALTER COLUMN col2 TYPE VARCHAR(64);
ERROR: ALTER TABLE ... ALTER COLUMN TYPE that rewrites table data may not affect
↳ replicated tables on a BDR node
```

To give an example with non-text types, consider col3 above with type INTEGER. An ATCT operation which tries to convert to SMALLINT or BIGINT will fail in a similar manner as above.

```
ALTER TABLE sample ALTER COLUMN col3 TYPE bigint;
ERROR: ALTER TABLE ... ALTER COLUMN TYPE that rewrites table data may not affect
↳ replicated tables on a BDR node
```

In both the above failing cases, there exists an automatic assignment cast from the current types to the target types. However there is no binary coercibility, which ends up causing a rewrite of the underlying table data.

In such cases, in controlled DBA environments, it is possible to change the type of a column to an automatically castable one, by adopting a rolling upgrade for the type of this column in a non-replicated environment on all the nodes, one by one. If the DDL is not replicated and the change of the column type is to an automatically castable one as above, then it is possible to allow the rewrite locally on the node performing the alter, along with concurrent activity on other nodes on this same table. This non-replicated ATCT operation can then be repeated on all the nodes one by one to bring about the desired change of the column type across the entire BDR cluster. Note that because this involves a rewrite, the activity will still take the DML lock for a brief period, and thus requires that the whole cluster is available. With the above specifics in place, the rolling upgrade of the non-replicated alter activity can be carried out as below:

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

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

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

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

```
SET bdr.ddl_replication TO FALSE;
-- conversion from BIGINT to TEXT works
ALTER TABLE sample ALTER COLUMN col3 TYPE TEXT;
-- conversion from TEXT back to BIGINT fails
ALTER TABLE sample ALTER COLUMN col3 TYPE BIGINT;
ERROR: ALTER TABLE ... ALTER COLUMN TYPE which cannot be automatically cast to new type
↳ may not affect replicated tables on a BDR node
RESET bdr.ddl_replication;
```

While the ATCT operations in non-replicated environments support a variety of type conversions, it is important to note that the rewrite can still fail if the underlying table data contains values that cannot be assigned to the new data type. For example, the current type for a column might be VARCHAR(256) and we tried a non-replicated ATCT operation to convert it into VARCHAR(128). If there is any existing data in the table which is wider than 128 bytes, then the rewrite operation will fail locally.

```
INSERT INTO sample VALUES (1, repeat('a', 200), 10);
SET bdr.ddl_replication TO FALSE;
ALTER TABLE sample ALTER COLUMN col2 TYPE VARCHAR(128);
INFO: in rewrite
ERROR: value too long for type character varying(128)
```

If underlying table data meets the characteristics of the new type, then the rewrite will succeed. However, there is a possibility that replication will fail if other nodes (which have not yet performed the non-replicated rolling data type upgrade) introduce new data that is wider than 128 bytes concurrently to this local ATCT operation. This will bring replication to a halt in the cluster. So it is important to be aware of the data type restrictions and characteristics at the database and application levels while performing these non-replicated rolling data type upgrade operations. It is **strongly** recommended and advisable to perform and test such ATCT operations in controlled and fully-aware DBA environments. We need to be aware that these ATCT operations are asymmetric, and backing out certain changes that fail could lead to table rewrites lasting long durations.

Also note that the above implicit castable ALTER activity cannot be performed in transaction blocks.

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

5.7.3 ALTER TYPE

Users should note that ALTER TYPE is replicated but a Global DML lock is *not* applied to all tables that use that data type, since PostgreSQL does not record those dependencies. See workarounds, below.

5.7.4 COMMENT ON

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

5.7.5 CREATE SEQUENCE

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

5.7.6 CREATE TABLE

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

5.7.7 CREATE TABLE AS and SELECT INTO

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

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

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

5.7.8 EXPLAIN

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

EXPLAIN ANALYZE Replication

EXPLAIN ANALYZE will follow replication rules of the analyzed statement.

EXPLAIN ANALYZE Locking

EXPLAIN ANALYZE will follow locking rules of the analyzed statement.

5.7.9 GRANT and REVOKE

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

5.7.10 LOCK TABLE

LOCK TABLE is only executed locally and is not replicated. Normal replication happens after transaction commit, so LOCK TABLE would not have any effect on other nodes.

For globally locking table, users can request a global DML lock explicitly by calling `bdr.global_lock_table()`.

5.7.11 SECURITY LABEL

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

5.7.12 TRUNCATE Replication

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

5.7.13 TRUNCATE Locking

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

5.7.14 Role manipulation statements

Users are global objects in a PostgreSQL instance, which means they span multiple databases while BDR operates on an individual database level. This means that role manipulation statement handling needs extra thought.

BDR requires that any roles that are referenced by any replicated DDL must exist on all nodes. The roles are not required to have the same grants, password, etc., but they must exist.

BDR will replicate role manipulation statements if `bdr.role_replication` is enabled (default) *and role manipulation statements are run in a BDR-enabled database.*

The role manipulation statements include the following statements:

- CREATE ROLE
- ALTER ROLE
- DROP ROLE
- GRANT ROLE
- CREATE USER

- ALTER USER
- DROP USER
- CREATE GROUP
- ALTER GROUP
- DROP GROUP

In general, either:

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

It is strongly recommended that you run all role management commands within one database.

If role replication is turned off, then the administrator must ensure that any roles used by DDL on one node also exist on the other nodes, or BDR apply will stall with an ERROR until the role is created on the other node(s).

Note: BDR will *not* capture and replicate role management statements when they are run on a non-BDR-enabled database within a BDR-enabled PostgreSQL instance. For example if you have DBs 'bdrdb' (bdr group member) and 'postgres' (bare db), and `bdr.role_replication = on`, then a CREATE USER run in bdrdb will be replicated, but a CREATE USER run in postgres will not.

5.7.15 Restricted DDL Workarounds

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

Adding a CONSTRAINT

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

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

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

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

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

Adding a Column

To add a column with a volatile default, run these commands in separate transactions:

```
ALTER TABLE mytable ADD COLUMN newcolumn coltype; -- Note the lack of DEFAULT or NOT NULL

ALTER TABLE mytable ALTER COLUMN newcolumn DEFAULT volatile-expression;

BEGIN;
SELECT bdr.global_lock_table('mytable');
UPDATE mytable SET newcolumn = default-expression;
COMMIT;
```

This splits schema changes and row changes into separate transactions that can be executed by BDR and result in consistent data across all nodes in a BDR group.

For best results, batch the update into chunks so that you do not update more than a few tens or hundreds of thousands of rows at once. This can be done using a PROCEDURE with embedded transactions.

It is important that the last batch of changes runs in a transaction that takes a global DML lock on the table, otherwise it is possible to miss rows that are inserted concurrently into the table on other nodes.

If required, `ALTER TABLE mytable ALTER COLUMN newcolumn NOT NULL;` can be run after the UPDATE has finished.

Changing a Column's Type

PostgreSQL causes a table rewrite in some cases where it could be avoided, for example:

```
CREATE TABLE foo (id BIGINT PRIMARY KEY, description VARCHAR(128));
ALTER TABLE foo ALTER COLUMN description TYPE VARCHAR(20);
```

This statement can be rewritten to avoid a table rewrite by making the restriction a table constraint rather than a datatype change, which can then be validated in a subsequent command to avoid long locks, if desired.

```
CREATE TABLE foo (id BIGINT PRIMARY KEY, description VARCHAR(128));
ALTER TABLE foo
  ALTER COLUMN description TYPE varchar,
  ADD CONSTRAINT description_length_limit CHECK (length(description) <= 20) NOT VALID;
ALTER TABLE foo VALIDATE CONSTRAINT description_length_limit;
```

Should the validation fail, then it is possible to UPDATE just the failing rows. This technique can be used for TEXT and VARCHAR using `length()`, or with NUMERIC datatype using `scale()`.

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

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

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

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

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

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

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

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

Changing Other Types

The ALTER TYPE statement is replicated, but affected tables are not locked.

When this DDL is used, the user should ensure that the statement has successfully executed on all nodes before using the new type. This can be achieved using the bdr.wait_slot_confirm_lsn() function.

For example,

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

will ensure that the DDL has been written to all nodes before using the new value in DML statements.

5.7.16 BDR Functions that behave like DDL

The following BDR management functions act like DDL. This means that they will attempt to take global locks and their actions will be replicated, if DDL replication is active and DDL filter settings allow that. For detailed information, see the documentation for the individual functions.

Replication Set Management

- bdr.create_replication_set
- bdr.alter_replication_set
- bdr.drop_replication_set
- bdr.replication_set_add_table
- bdr.replication_set_remove_table
- bdr.replication_set_add_ddl_filter
- bdr.replication_set_remove_ddl_filter

Conflict Management

- bdr.alter_table_conflict_detection
- bdr.column_timestamps_enable
- bdr.column_timestamps_disable

Sequence Management

- bdr.alter_sequence_set_kind

Stream Triggers

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

SECURITY AND ROLES

The BDR3 extension can be created only by superusers, although if desired, it is possible to set up the `pgextwlist` extension and configure it to allow BDR3 to be created by a non-superuser.

Configuring and managing BDR3 does not require superuser access, nor is that recommended. The privileges required by BDR3 are split across the following default/predefined roles, named similarly to the PostgreSQL default/predefined roles:

- *bdr_superuser* - the highest-privileged role, having access to all BDR tables and functions.
- *bdr_read_all_stats* - the role having read-only access to the tables, views and functions, sufficient to understand the state of BDR.
- *bdr_monitor* - at the moment the same as *bdr_read_all_stats*, to be extended later.
- *bdr_application* - the minimal privileges required by applications running BDR.
- *bdr_read_all_conflicts* - can view *all* conflicts in `bdr.conflict_log`.

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

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

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

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

6.1 Catalog Tables

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

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

6.2 BDR Functions & Operators

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

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

6.3 Granting privileges on catalog objects

Administrators should not grant explicit privileges on catalog objects such as tables, views and functions; manage access to those objects by granting one of the roles documented in [BDR Default Roles].

This requirement is a consequence of the flexibility that allows joining a node group even if the nodes on either side of the join do not have the exact same version of BDR (and therefore of the BDR catalog).

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

6.4 Role Management

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

When a new BDR node joins the BDR group, existing users are not automatically copied unless the node is added using `bdr_init_physical`. This is intentional and is an important security feature. PostgreSQL allows users to access multiple databases, with the default being to access any database. BDR does not know which users access which database and so cannot safely decide which users to copy across to the new node.

PostgreSQL allows you to dump all users with the command:

```
pg_dumpall --roles-only > roles.sql
```

The file `roles.sql` can then be edited to remove unwanted users before re-executing that on the newly created node. Other mechanisms are possible, depending on your identity and access management solution (IAM), but are not automated at this time.

6.5 Roles and Replication

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

DML changes to tables are replicated as the table-owning user on the target node. It is recommended - but not enforced - that a table is owned by the same user on each node.

If table A is owned by user X on node1 and owned by user Y on node2, then if user Y has higher privileges than user X, this could be viewed as a privilege escalation. Since some nodes have different use cases, we allow this but warn against it to allow the security administrator to plan and audit this situation.

On tables with row level security policies enabled, changes will be replicated without re-enforcing policies on apply. This is equivalent to the changes being applied as `NO FORCE ROW LEVEL SECURITY`, even if `FORCE ROW LEVEL SECURITY` is specified. If this is not desirable, specify a `row_filter` that avoids replicating all rows. It is recommended - but not enforced - that the row security policies on all nodes be identical or at least compatible.

Note that `bdr_superuser` controls replication for BDR and may add/remove any table from any replication set. `bdr_superuser` does not need, nor is it recommended to have, any privileges over individual tables. If the need exists to restrict access to replication set functions, restricted versions of these functions can be implemented as `SECURITY DEFINER` functions and GRANTED to the appropriate users.

6.6 Connection Role

When allocating a new BDR node, the user supplied in the DSN for the `local_dsn` argument of `bdr.create_node` and the `join_target_dsn` of `bdr.join_node_group` are used frequently to refer to, create, and manage database objects. This is especially relevant during the initial bootstrapping process, where the specified accounts may invoke operations on database objects directly or through the `pglogical` module rather than BDR.

BDR is carefully written to prevent privilege escalation attacks even when using a role with `SUPERUSER` rights in these DSNs.

To further reduce the attack surface, a more restricted user may be specified in the above DSNs. At a minimum, such a user must be granted permissions on all nodes, such that following stipulations are satisfied:

- the user has the `REPLICATION` attribute
- it is granted the `CREATE` permission on the database
- it inherits the `pglogical_superuser` and `bdr_superuser` roles
- it owns all database objects to replicate, either directly or via permissions from the owner role(s).

Once all nodes are joined, the permissions may be further reduced to just the following to still allow DML and DDL replication:

- The user has the `REPLICATION` attribute.
- It inherits the `pglogical_superuser` and `bdr_superuser` roles.

6.7 Privilege Restrictions

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

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

6.7.1 Foreign Key Privileges

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

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

6.7.2 Triggers

In PostgreSQL, triggers may be created by both the owner of a table and anyone who has been granted the `TRIGGER` privilege. Triggers granted by the non-table owner would execute as the table owner in BDR, which could cause a security issue. The `TRIGGER` privilege is seldom used and PostgreSQL Core Team has said “The separate `TRIGGER` permission is something we consider obsolescent.”

BDR mitigates this problem by using stricter rules on who can create a trigger on a table:

- `superuser`
- `bdr_superuser`

- Owner of the table can create triggers according to same rules as in PostgreSQL (must have EXECUTE privilege on function used by the trigger).
- Users who have TRIGGER privilege on the table can only create a trigger if they create the trigger using a function that is owned by the same owner as the table and they satisfy standard PostgreSQL rules (again must have EXECUTE privilege on the function). So if both table and function have same owner and the owner decided to give a user both TRIGGER privilege on the table and EXECUTE privilege on the function, it is assumed that it is okay for that user to create a trigger on that table using this function.
- Users who have TRIGGER privilege on the table can create triggers using functions that are defined with the SECURITY DEFINER clause if they have EXECUTE privilege on them. This clause makes the function always execute in the context of the owner of the function itself both in standard PostgreSQL and BDR.

The above logic is built on the fact that in PostgreSQL, the owner of the trigger is not the user who created it but the owner of the function used by that trigger.

The same rules apply to existing tables, and if the existing table has triggers which are not owned by the owner of the table and do not use SECURITY DEFINER functions, it will not be possible to add it to a replication set.

These checks were added with BDR 3.6.19. An application that relies on the behavior of previous versions can set `bdr.backwards_compatibility` to 30618 (or lower) to behave like earlier versions.

BDR replication apply uses the system-level default `search_path` only. Replica triggers, stream triggers and index expression functions may assume other `search_path` settings which will then fail when they execute on apply. To ensure this does not occur, resolve object references clearly using either the default `search_path` only (always use fully qualified references to objects, e.g. `schema.objectname`), or set the search path for a function using `ALTER FUNCTION ... SET search_path = ...` for the functions affected.

6.8 BDR Default/Predefined Roles

BDR predefined roles are created when the BDR3 extension is installed. Note that after BDR3 extension is dropped from a database, the roles continue to exist and need to be dropped manually if required. This allows BDR to be used in multiple databases on the same PostgreSQL instance without problem.

Remember that the `GRANT ROLE` DDL statement does not participate in BDR replication, thus you should execute this on each node of a cluster.

6.8.1 `bdr_superuser`

- ALL PRIVILEGES ON ALL TABLES IN SCHEMA BDR
- ALL PRIVILEGES ON ALL ROUTINES IN SCHEMA BDR

6.8.2 `bdr_read_all_stats`

SELECT privilege on

- `bdr.conflict_history_summary`
- `bdr.ddl_epoch`
- `bdr.ddl_replication`
- `bdr.global_consensus_journal_details`
- `bdr.global_lock`
- `bdr.global_locks`

- `bdr.local_consensus_state`
- `bdr.local_node_summary`
- `bdr.node`
- `bdr.node_catchup_info`
- `bdr.node_conflict_resolvers`
- `bdr.node_group`
- `bdr.node_local_info`
- `bdr.node_peer_progress`
- `bdr.node_slots`
- `bdr.node_summary`
- `bdr.replication_sets`
- `bdr.sequences`
- `bdr.state_journal_details`
- `bdr.stat_relation`
- `bdr.stat_subscription`
- `bdr.subscription`
- `bdr.subscription_summary`
- `bdr.tables`
- `bdr.worker_errors`

EXECUTE privilege on

- `bdr.bdr_edition`
- `bdr.bdr_version`
- `bdr.bdr_version_num`
- `bdr.conflict_resolution_to_string`
- `bdr.conflict_type_to_string`
- `bdr.decode_message_payload`
- `bdr.get_global_locks`
- `bdr.get_raft_status`
- `bdr.get_relation_stats`
- `bdr.get_slot_flush_timestamp`
- `bdr.get_sub_progress_timestamp`
- `bdr.get_subscription_stats`
- `bdr.peer_state_name`
- `bdr.show_subscription_status`

6.8.3 bdr_monitor

All privileges from `bdr_read_all_stats`, plus
EXECUTE privilege on

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

6.8.4 bdr_application

EXECUTE privilege on

- All functions for `column_timestamps` datatypes
- All functions for `CRDT` datatypes
- `bdr.alter_sequence_set_kind`
- `bdr.create_conflict_trigger`
- `bdr.create_transform_trigger`
- `bdr.drop_trigger`
- `bdr.get_configured_camo_partner_of`
- `bdr.get_configured_camo_origin_for`
- `bdr.global_lock_table`
- `bdr.is_camo_partner_connected`
- `bdr.is_camo_partner_ready`
- `bdr.logical_transaction_status`
- `bdr.ri_fkey_trigger`
- `bdr.seq_nextval`
- `bdr.seq_currval`
- `bdr.seq_lastval`
- `bdr.trigger_get_committs`
- `bdr.trigger_get_conflict_type`
- `bdr.trigger_get_origin_node_id`
- `bdr.trigger_get_row`
- `bdr.trigger_get_type`
- `bdr.trigger_get_xid`
- `bdr.wait_for_camo_partner_queue`
- `bdr.wait_slot_confirm_lsn`

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

6.8.5 bdr_read_all_conflicts

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

6.9 Verification

BDR has been verified using the following tools and approaches.

6.9.1 Coverity

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

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

6.9.2 CIS Benchmark

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

	R esult	Description
1 .4	PASS	Ensure systemd Service Files Are Enabled
1 .5	PASS	Ensure Data Cluster Initialized Successfully
2 .1	PASS	Ensure the file permissions mask is correct
2 .2	PASS	Ensure the PostgreSQL pg_wheel group membership is correct
3 .1 .2	PASS	Ensure the log destinations are set correctly
3 .1 .3	PASS	Ensure the logging collector is enabled
3 .1 .4	PASS	Ensure the log file destination directory is set correctly
3 .1 .5	PASS	Ensure the filename pattern for log files is set correctly
3 .1 .6	PASS	Ensure the log file permissions are set correctly
3 .1 .7	PASS	Ensure 'log_truncate_on_rotation' is enabled
3 .1 .8	PASS	Ensure the maximum log file lifetime is set correctly
3 .1 .9	PASS	Ensure the maximum log file size is set correctly
3 .1 .10	PASS	Ensure the correct syslog facility is selected
3 .1 .11	PASS	Ensure the program name for PostgreSQL syslog messages is correct
3 .1 .14	PASS	Ensure 'debug_print_parse' is disabled
3 .1 .15	PASS	Ensure 'debug_print_rewritten' is disabled
3 .1 .16	PASS	Ensure 'debug_print_plan' is disabled

continues on next page

Table 1 – continued from previous page

	R esult	Description
3. 1. 17	PASS	Ensure 'debug_pretty_print' is enabled
3. 1. 18	PASS	Ensure 'log_connections' is enabled
3. 1. 19	PASS	Ensure 'log_disconnections' is enabled
3. 1. 21	PASS	Ensure 'log_hostname' is set correctly
3. 1. 23	PASS	Ensure 'log_statement' is set correctly
3. 1. 24	PASS	Ensure 'log_timezone' is set correctly
3 .2	PASS	Ensure the PostgreSQL Audit Extension (pgAudit) is enabled
4 .1	PASS	Ensure sudo is configured correctly
4 .2	PASS	Ensure excessive administrative privileges are revoked
4 .3	PASS	Ensure excessive function privileges are revoked
4 .4	PASS	Tested Ensure excessive DML privileges are revoked
5 .2	Not T ested	Ensure login via 'host' TCP/IP Socket is configured correctly
6 .2	PASS	Ensure 'backend' runtime parameters are configured correctly
6 .7	Not T ested	Ensure FIPS 140-2 OpenSSL Cryptography Is Used
6 .8	PASS	Ensure SSL is enabled and configured correctly
7 .3	PASS	Ensure WAL archiving is configured and functional

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

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

CONFLICTS

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

Conflicts aren't **ERRORs** - they are events that can be detected and resolved automatically as they occur by BDR, in most cases. Resolution depends upon the nature of the application and the meaning of the data, so it is important that BDR provides the application a range of choices as to how to resolve conflicts.

By default, conflicts are resolved at row level. That is, when changes from two nodes conflict, we pick either the local or remote tuple and discard the other one. For example, we may compare commit timestamps for the two conflicting changes, and keep the newer one. This ensures that all nodes converge to the same result, and establishes commit-order-like semantics on the whole cluster.

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

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

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

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

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

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

Some clustering systems use distributed lock mechanisms to prevent concurrent access to data. These can perform reasonably when servers are very close, but cannot support geographically distributed applications where very low latency is critical for acceptable performance.

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

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

7.1 How conflicts happen

Inter-node conflicts arise as a result of sequences of events that could not happen if all the involved transactions happened concurrently on the same node. Because the nodes only exchange changes after the transactions commit, each transaction is individually valid on the node it committed on, but would not be valid if applied on another node that did other conflicting work at the same time.

Since BDR replication essentially replays the transaction on the other nodes, the replay operation can fail if there is a conflict between a transaction being applied and a transaction that was committed on the receiving node.

The reason most conflicts can't happen when all transactions run on a single node is that PostgreSQL has inter-transaction communication mechanisms to prevent it - `UNIQUE` indexes, `SEQUENCES`, row and relation locking, `SERIALIZABLE` dependency tracking, etc. All of these mechanisms are ways to communicate between ongoing transactions to prevent undesirable concurrency issues.

BDR does not have a distributed transaction manager or lock manager. That's part of why it performs well with latency and network partitions. As a result, *transactions on different nodes execute entirely independently from each other*, when using the default, lazy replication. Less independence between nodes can avoid conflicts altogether, which is why BDR also offers eager replication for when this is important.

7.2 Types of conflict

7.2.1 PRIMARY KEY or UNIQUE Conflicts

The most common conflicts are row conflicts, where two operations affect a row with the same key in ways they could not do on a single node. BDR can detect most of those and will apply the `update_if_newer` conflict resolver.

Row conflicts include:

- `INSERT` vs `INSERT`
- `UPDATE` vs `UPDATE`
- `UPDATE` vs `DELETE`
- `INSERT` vs `UPDATE`
- `INSERT` vs `DELETE`
- `DELETE` vs `DELETE`

The view `bdr.node_conflict_resolvers` provides information on how conflict resolution is currently configured for all known conflict types.

INSERT/INSERT Conflicts

The most common conflict, `INSERT/INSERT`, arises where `INSERT`s on two different nodes create a tuple with the same `PRIMARY KEY` values (or if no `PRIMARY KEY` exists, the same values for a single `UNIQUE` constraint).

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

This conflict will generate the `insert_exists` conflict type, which is by default resolved by choosing the newer (based on commit time) row and keeping only that one (`update_if_newer` resolver). Other resolvers can be configured - see [Conflict Resolution](#) for details.

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

This type of conflict can be effectively eliminated by use of [Global Sequences](#).

INSERTs that Violate Multiple UNIQUE Constraints

An INSERT/INSERT conflict can violate more than one UNIQUE constraint (of which one might be the PRIMARY KEY). If a new row violates more than one UNIQUE constraint and that results in a conflict against more than one other row, then the apply of the replication change will produce a `multiple_unique_conflicts` conflict.

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

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

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

UPDATE/UPDATE Conflicts

Where two concurrent UPDATES on different nodes change the same tuple (but not its PRIMARY KEY), an UPDATE/UPDATE conflict can occur on replay.

These can generate different conflict kinds based on the configuration and situation. If the table is configured with *Row Version Conflict Detection*, then the original (key) row is compared with the local row; if they are different, the `update_differing` conflict is generated. When using *Origin Conflict Detection*, the origin of the row is checked (the origin is the node that the current local row came from); if that has changed, the `update_origin_change` conflict is generated. In all other cases, the UPDATE is normally applied without a conflict being generated.

Both of these conflicts are resolved same way as `insert_exists`, as described above.

UPDATE Conflicts on the PRIMARY KEY

BDR cannot currently perform conflict resolution where the PRIMARY KEY is changed by an UPDATE operation. It is permissible to update the primary key, but you must ensure that no conflict with existing values is possible.

Conflicts on the update of the primary key are *Divergent Conflicts* and require manual operator intervention.

Updating a PK is possible in PostgreSQL, but there are issues in both PostgreSQL and BDR.

Let's create a very simple example schema to explain:

```
CREATE TABLE pktest (pk integer primary key, val integer);
INSERT INTO pktest VALUES (1,1);
```

Updating the Primary Key column is possible, so this SQL succeeds:

```
UPDATE pktest SET pk=2 WHERE pk=1;
```

...but if we have multiple rows in the table, e.g.:

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

...then some UPDATES would succeed:

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

```
SELECT * FROM pktest;
```

```
pk| val
---+---
 2|  1
 4|  3
(2 rows)
```

...but other UPDATES would fail with constraint errors:

```
UPDATE pktest SET pk=4 WHERE pk=2;
```

```
ERROR:  duplicate key value violates unique constraint "pktest_pkey"
```

```
DETAIL:  Key (pk)=(4) already exists
```

So for PostgreSQL applications that UPDATE PKs, be very careful to avoid runtime errors, even without BDR.

With BDR, the situation becomes more complex if UPDATES are allowed from multiple locations at same time.

Executing these two changes concurrently works:

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

```
node2: UPDATE pktest SET pk=pk+1 WHERE pk = 4;
```

```
SELECT * FROM pktest;
```

```
pk| val
---+---
 3|  1
 5|  3
(2 rows)
```

...but executing these next two changes concurrently will cause a divergent error, since both changes are accepted. But when the changes are applied on the other node, this will result in update_missing conflicts.

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

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

...leaving the data different on each node:

```
node1:
```

```
SELECT * FROM pktest;
```

```
pk| val
---+---
 1|  1
 5|  3
(2 rows)
```

```
node2:
```

```
SELECT * FROM pktest;
```

```
pk| val
---+---
 2|  1
 5|  3
(2 rows)
```

This situation can be identified and resolved using LiveCompare.

Concurrent conflicts give problems. Executing these two changes concurrently is not easily resolvable:

```
node1: UPDATE pktest SET pk=6, val=8 WHERE pk = 5;
node2: UPDATE pktest SET pk=6, val=9 WHERE pk = 5;
```

Both changes are applied locally, causing a divergence between the nodes. But then apply on the target fails on both nodes with a duplicate key value violation ERROR, which causes the replication to halt and currently requires manual resolution.

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

You can avoid divergence in cases like the one described above where the same old key is being updated by the same new key concurrently by setting `update_pkey_exists` to `update_if_newer`. However in certain situations, divergence will happen even with `update_if_newer`, namely when 2 different rows both get updated concurrently to the same new primary key.

As a result, we recommend strongly against allowing PK UPDATES in your applications, especially with BDR. If there are parts of your application that change Primary Keys, then to avoid concurrent changes, make those changes using Eager replication.

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

UPDATES that Violate Multiple UNIQUE Constraints

Like *INSERTs that Violate Multiple UNIQUE Constraints*, where an incoming UPDATE violates more than one UNIQUE index (and/or the PRIMARY KEY), BDR will raise a `multiple_unique_conflicts` conflict.

BDR supports deferred unique constraints. If a transaction can commit on the source then it will apply cleanly on target, unless it sees conflicts. However, a deferred Primary Key cannot be used as a REPLICA IDENTITY, so the use cases are already limited by that and the warning above about using multiple unique constraints.

UPDATE/DELETE Conflicts

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

If the DELETED row is still detectable (the deleted row wasn't removed by VACUUM), the `update_recently_deleted` conflict will be generated. By default the UPDATE will just be skipped, but the resolution for this can be configured; see *Conflict Resolution* for details.

The deleted row can be cleaned up from the database by the time the UPDATE is received in case the local node is lagging behind in replication. In this case BDR cannot differentiate between UPDATE/DELETE conflicts and *INSERT/UPDATE Conflicts* and will simply generate the `update_missing` conflict.

Another type of conflicting DELETE and UPDATE is a DELETE operation that comes after the row was UPDATED locally. In this situation, the outcome depends upon the type of conflict detection used. When using the default, *Origin Conflict Detection*, no conflict is detected at all, leading to the DELETE being applied and the row removed. If you enable *Row Version Conflict Detection*, a `delete_recently_updated` conflict is generated. The default resolution for this conflict type is to apply the DELETE and remove the row, but this can be configured or handled via a conflict trigger.

INSERT/UPDATE Conflicts

When using the default asynchronous mode of operation, a node may receive an UPDATE of a row before the original INSERT was received. This can only happen with 3 or more nodes being active (see *Conflicts with 3 or more nodes* below).

When this happens, the `update_missing` conflict is generated. The default conflict resolver is `insert_or_skip`, though `insert_or_error` or `skip` may be used instead. Resolvers that do insert-or-action will first try to INSERT a new row based on data from the UPDATE when possible (when the whole row was received). For the reconstruction of the row to be possible, the table either needs to have `REPLICA IDENTITY FULL` or the row must not contain any TOASTed data.

See *TOAST Support Details* for more info about TOASTed data.

INSERT/DELETE Conflicts

Similarly to the INSERT/UPDATE conflict, the node may also receive a DELETE operation on a row for which it didn't receive an INSERT yet. This is again only possible with 3 or more nodes set up (see *Conflicts with 3 or more nodes* below).

BDR cannot currently detect this conflict type: the INSERT operation will not generate any conflict type and the INSERT will be applied.

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

DELETE/DELETE Conflicts

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

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

This conflict is harmless since both DELETES have the same effect, so one of them can be safely ignored.

Conflicts with 3 or more nodes

If one node INSERTs a row which is then replayed to a 2nd node and UPDATED there, a 3rd node can receive the UPDATE from the 2nd node before it receives the INSERT from the 1st node. This is an INSERT/UPDATE conflict.

These conflicts are handled by discarding the UPDATE. This can lead to *different data on different nodes*, i.e. these are *Divergent Conflicts*.

Note that this conflict type can only happen with 3 or more masters, of which at least 2 must be actively writing.

Also, the replication lag from node 1 to node 3 must be high enough to allow the following sequence of actions:

1. node 2 receives INSERT from node 1
2. node 2 performs UPDATE
3. node 3 receives UPDATE from node 2
4. node 3 receives INSERT from node 1

Using `insert_or_error` (or in some cases the `insert_or_skip` conflict resolver for the `update_missing` conflict type) is a viable mitigation strategy for these conflicts. Note however that enabling this option opens the door for INSERT/DELETE conflicts; see below.

1. node 1 performs UPDATE

2. node 2 performs DELETE
3. node 3 receives DELETE from node 2
4. node 3 receives UPDATE from node 1, turning it into an INSERT

If these are problems, it's recommended to tune freezing settings for a table or database so that they are correctly detected as `update_recently_deleted`.

Another alternative is to use [Eager Replication] to prevent these conflicts.

INSERT/DELETE conflicts can also occur with 3 or more nodes. Such a conflict is identical to INSERT/UPDATE, except with the UPDATE replaced by a DELETE. This can result in a `delete_missing` conflict.

BDR could choose to make each INSERT into a check-for-recently deleted, as occurs with an `update_missing` conflict. However, the cost of doing this penalizes the majority of users, so at this time we simply log `delete_missing`.

Later releases will automatically resolve INSERT/DELETE anomalies via re-checks using LiveCompare when `delete_missing` conflicts occur. These can be performed manually by applications by checking conflict logs or conflict log tables; see later.

These conflicts can occur in two main problem use cases:

- INSERT, followed rapidly by a DELETE - as can be used in queuing applications
- Any case where the PK identifier of a table is re-used

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

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

Applications that insert, delete and then later re-use the same unique identifiers can cause difficulties. This is known as the ABA Problem. BDR has no way of knowing whether the rows are the current row, the last row or much older rows. https://en.wikipedia.org/wiki/ABA_problem

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

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

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

Any application that uses Sequences or UUIDs will not suffer from this problem.

7.2.2 Foreign Key Constraint Conflicts

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

BDR applies changes with `session_replication_role = 'replica'`, so foreign keys are **not** re-checked when applying changes. In an active/active environment this can result in FK violations if deletes occur to the referenced table at the same time as inserts into the referencing table. This is similar to an INSERT/DELETE conflict.

First we will explain the problem, and then provide solutions.

In single-master PostgreSQL, any INSERT/UPDATE that refers to a value in the referenced table will have to wait for DELETES to finish before they can gain a row-level lock. If a DELETE removes a referenced value, then the INSERT/UPDATE will fail the FK check.

In multi-master BDR there are no inter-node row-level locks. So an INSERT on the referencing table does not wait behind a DELETE on the referenced table, so both actions can occur concurrently. Thus an INSERT/UPDATE on one node on the referencing table can utilize a value at the same time as a DELETE on the referenced table on another node. This then results in a value in the referencing table that is no longer present in the referenced table.

In practice, this only occurs if DELETES occur on referenced tables in separate transactions from DELETES on referencing tables. This is not a common operation.

In a parent-child relationship, e.g. Orders -> OrderItems, it isn't typical to do this; it is more likely to mark an OrderItem as cancelled than to remove it completely. For reference/lookup data, it would be strange to completely remove entries at the same time as using those same values for new fact data.

While there is a possibility of dangling FKs, the risk of this in general is very low and so BDR does not impose a generic solution to cover this case. Once users understand the situation in which this occurs, two solutions are possible:

The first solution is to restrict the use of FKs to closely related entities that are generally modified from only one node at a time, are infrequently modified, or where the modification's concurrency is application-mediated. This simply avoids any FK violations at the application level.

The second solution is to add triggers to protect against this case using the BDR-provided functions `bdr.ri_fkey_trigger()` and `bdr.ri_fkey_on_del_trigger()`. When called as BEFORE triggers, these functions will use FOREIGN KEY information to avoid FK anomalies by setting referencing columns to NULL, much as if we had a SET NULL constraint. Note that this re-checks ALL FKs in one trigger, so you only need to add one trigger per table to prevent FK violation.

As an example, we have two tables: Fact and RefData. Fact has an FK that references RefData. Fact is the referencing table and RefData is the referenced table. One trigger needs to be added to each table.

Add a trigger that will set columns to NULL in Fact if the referenced row in RefData has already been deleted.

```
CREATE TRIGGER bdr_replica_fk_iu_trg
  BEFORE INSERT OR UPDATE ON fact
  FOR EACH ROW
  EXECUTE PROCEDURE bdr.ri_fkey_trigger();

ALTER TABLE fact
  ENABLE REPLICA TRIGGER bdr_replica_fk_iu_trg;
```

Add a trigger that will set columns to NULL in Fact at the time a DELETE occurs on the RefData table.

```
CREATE TRIGGER bdr_replica_fk_d_trg
  BEFORE DELETE ON refdata
  FOR EACH ROW
  EXECUTE PROCEDURE bdr.ri_fkey_on_del_trigger();

ALTER TABLE refdata
  ENABLE REPLICA TRIGGER bdr_replica_fk_d_trg;
```

Adding both triggers will avoid dangling foreign keys.

7.2.3 TRUNCATE Conflicts

TRUNCATE behaves similarly to a DELETE of all rows, but performs this action by physical removal of the table data, rather than row-by-row deletion. As a result, row-level conflict handling is not available, so TRUNCATE commands do not generate conflicts with other DML actions, even when there is a clear conflict.

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

Users may wish to take one of the following actions:

- Ensure TRUNCATE is not executed alongside other concurrent DML and rely on LiveCompare to highlight any such inconsistency.
- Replace TRUNCATE with a DELETE statement with no WHERE clause, noting that this is likely to have very poor performance on larger tables.
- Set `bdr.truncate_locking = 'on'` to set the TRUNCATE command's locking behavior. Determines whether TRUNCATE obeys the `bdr.ddl_locking` setting. This is not the default behaviour for TRUNCATE since it requires all nodes to be up, so may not be possible or desirable in all cases.

7.2.4 Exclusion Constraint Conflicts

BDR does not support exclusion constraints, and prevents their creation.

If an existing stand-alone database is converted to a BDR database then all exclusion constraints should be manually dropped.

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

If you force BDR to create an exclusion constraint, or you do not drop existing ones when converting a standalone database to BDR, you should expect replication to break. To get it to progress again, remove or alter the local tuple(s) that an incoming remote tuple conflicts with, so that the remote transaction can be applied.

7.2.5 Data Conflicts for Roles and Tablespace differences

Conflicts can also arise where nodes have global (PostgreSQL-system-wide) data, like roles, that differ. This can result in operations - mainly DDL - that can be run successfully and committed on one node, but then fail to apply to other nodes.

For example, `node1` might have a user named `fred`, but that user was not created on `node2`. If `fred` on `node1` creates a table, it will be replicated with its owner set to `fred`. When the DDL command is applied to `node2`, the DDL will fail because there is no user named `fred`. This failure will emit an ERROR in the PostgreSQL logs.

Administrator intervention is required to resolve this conflict by creating the user `fred` in the database where BDR is running. You may wish to set `bdr.role_replication = on` to resolve this in future.

7.2.6 Lock Conflicts and Deadlock Aborts

Because BDR writer processes operate much like normal user sessions, they are subject to the usual rules around row and table locking. This can sometimes lead to BDR writer processes waiting on locks held by user transactions, or even by each other.

Relevant locking includes:

- explicit table-level locking (`LOCK TABLE ...`) by user sessions
- explicit row-level locking (`SELECT ... FOR UPDATE/FOR SHARE`) by user sessions
- implicit locking because of row `UPDATES`, `INSERTs` or `DELETEs`, either from local activity or from replication from other nodes

It is even possible for a BDR writer process to deadlock with a user transaction, where the user transaction is waiting on a lock held by the writer process, and vice versa. Two writer processes may also deadlock with each other. PostgreSQL's deadlock detector will step in and terminate one of the problem transactions. If the BDR writer process is terminated, it will simply retry, and generally succeed.

All these issues are transient and generally require no administrator action. If a writer process is stuck for a long time behind a lock on an idle user session, the administrator may choose to terminate the user session to get replication flowing again, but this is no different to a user holding a long lock that impacts another user session.

Use of the `log_lock_waits` facility in PostgreSQL can help identify locking related replay stalls.

7.2.7 Divergent Conflicts

Divergent conflicts arise when data that should be the same on different nodes differs unexpectedly. Divergent conflicts should not occur, but not all such conflicts can be reliably prevented at the time of writing.

Changing the `PRIMARY KEY` of a row can lead to a divergent conflict if another node changes the key of the same row before all nodes have replayed the change. Avoid changing primary keys, or change them only on one designated node.

Divergent conflicts involving row data generally require administrator action to manually adjust the data on one of the nodes to be consistent with the other one. Such conflicts should not arise so long as BDR is used as documented, and settings or functions marked as unsafe are avoided.

The administrator must manually resolve such conflicts. Use of the advanced options such as `bdr.ddl_replication` and `bdr.ddl_locking` may be required depending on the nature of the conflict. However, careless use of these options can make things much worse and it is not possible to give general instructions for resolving all possible kinds of conflict.

7.2.8 TOAST Support Details

PostgreSQL uses out of line storage for larger columns called `TOAST`.

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

The `TOAST` value will be logged into the transaction log (WAL) only if the value has changed. This can cause problems, especially when handling `UPDATE` conflicts because an `UPDATE` statement that did not change a value of a toasted column will produce a row without that column. As mentioned in *INSERT/UPDATE Conflicts*, BDR will produce an error if an `update_missing` conflict is resolved using `insert_or_error` and there are missing `TOAST` columns.

However, there are more subtle issues than the above one in case of concurrent workloads with asynchronous replication (eager transactions are not affected). Imagine for example the following workload on a BDR cluster with 3 nodes called A, B and C:

1. on node A: txn A1 does an `UPDATE SET col1 = 'toast data...'` and commits first

2. on node B: txn B1 does UPDATE SET other_column = 'anything else'; and commits after A1
3. on node C: the connection to node A lags behind
4. on node C: txn B1 is applied first, it misses the TOASTed column in col1, but gets applied without conflict
5. on node C: txn A1 will conflict (on update_origin_change) and get skipped
6. node C will miss the toasted data from A1 forever

The above is not usually a problem when using BDR (it would be when using either built-in logical replication or plain pglogical for multi-master) because BDR adds its own logging of TOAST columns when it detects a local UPDATE to a row which recently replicated a TOAST column modification, and the local UPDATE is not modifying the TOAST. Thus BDR will prevent any inconsistency for TOASTed data across different nodes, at the price of increased WAL logging when updates occur on multiple nodes (i.e. when origin changes for a tuple). Additional WAL overhead will be zero if all updates are made from a single node, as is normally the case with BDR AlwaysOn architecture.

!!! Note Running VACUUM FULL or CLUSTER on just the TOAST table without also doing same on the main table will remove metadata needed for the extra logging to work, which means that, for a short period of time after such a statement, the protection against these concurrency issues will not be present.

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

For the insert_or_error conflict resolution, the use of REPLICA IDENTITY FULL is however still required.

None of these problems associated with TOASTed columns affect tables with REPLICA IDENTITY FULL as this setting will always log a TOASTed value as part of the key since the whole row is considered to be part of the key. Both BDR and pglogical are smart enough to reconstruct the new row, filling the missing data from the key row. Be aware that as a result, the use of REPLICA IDENTITY FULL can increase WAL size significantly.

7.3 Avoiding or Tolerating Conflicts

In most cases the application can be designed to avoid conflicts, or to tolerate them.

Conflicts can only happen if there are things happening at the same time on multiple nodes, so the simplest way to avoid conflicts is to only ever write to one node, or to only ever write to a specific row in a specific way from one specific node at a time.

This happens naturally in many applications. For example, many consumer applications only allow data to be changed by the owning user, e.g. changing the default billing address on your account, so data changes seldom experience update conflicts.

It might happen that you make a change just before a node goes down, so the change appears to have been lost. You might then make the same change again, leading to two updates via different nodes. When the down node comes back up, it will try to send the older change to other nodes, but it will be rejected because the last update of the data is kept.

For INSERT/INSERT conflicts, use of [Global Sequences](#) can completely prevent this type of conflict.

For applications that assign relationships between objects, e.g. a room booking application, applying update_if_newer may not give an acceptable business outcome, i.e. it isn't useful to confirm to two people separately that they have booked the same room. The simplest resolution is to use Eager replication to ensure that only one booking succeeds. More complex ways might be possible depending upon the application, e.g. assign 100 seats to each node and allow those to be booked by a writer on that node, but if none are available locally, use a distributed locking scheme or Eager replication once most seats have been reserved.

Another technique for ensuring certain types of update only occur from one specific node would be to route different types of transaction through different nodes. For example:

- receiving parcels on one node, but delivering parcels via another node.
- a service application where orders are input on one node, work is prepared on a second node and then served back to customers on another.

The best course of action is frequently to allow conflicts to occur and design the application to work with BDR's conflict resolution mechanisms to cope with the conflict.

7.4 Conflict Detection

BDR provides these mechanisms for conflict detection:

- *Origin Conflict Detection* (default)
- *Row Version Conflict Detection*
- *Column-Level Conflict Detection*

7.4.1 Origin Conflict Detection

(Previously known as Timestamp Conflict Detection, but this was confusing.)

Origin conflict detection uses and relies on commit timestamps as recorded on the host where the transaction originates from. This requires clocks to be in sync to work correctly, or to be within a tolerance of the fastest message between two nodes. If this is not the case, conflict resolution will tend to favour the node that is further ahead. Clock skew between nodes can be managed using the parameters `bdr.maximum_clock_skew` and `bdr.maximum_clock_skew_action`.

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

Conflicts are initially detected based upon whether the replication origin has changed or not, so conflict triggers will be called in situations that may turn out not to be actual conflicts. Hence, this mechanism is not precise since it can generate false positive conflicts.

Origin info is available only up to the point where a row is frozen. Updates arriving for a row after it has been frozen will not raise a conflict, so will be applied in all cases. This is the normal case when we add a new node by `bdr_init_physical`, so raising conflicts would cause many false positive cases in that case.

When a node that has been offline for some time reconnects and begins sending data changes, this could potentially cause divergent errors if the newly arrived updates are actually older than the frozen rows that they update. Inserts and Deletes are not affected by this situation.

Users are advised to not leave down nodes for extended outages, as discussed in [Node Restart and Down Node Recovery](#).

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

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

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

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

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

Initially recommended settings would be:

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

vacuum_freeze_min_age = 500000000

# 90% of autovacuum_freeze_max_age
vacuum_freeze_table_age = 900000000
```

Note that:

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

7.4.2 Row Version Conflict Detection

Alternatively, BDR provides the option to use row versioning and make conflict detection independent of the nodes' system clock.

Row version conflict detection requires 3 things to be enabled. If any of these steps are not performed correctly then *Origin Conflict Detection* will be used.

1. `check_full_tuple` must be enabled for the BDR node group.
2. `REPLICA IDENTITY FULL` must be enabled on all tables that are to use row version conflict detection.
3. Row Version Tracking must be enabled on the table by using `bdr.alter_table_conflict_detection`. This function will add a new column (with a user defined name) and an `UPDATE` trigger which manages the new column value. The column will be created as `INTEGER` type.

Although the counter is incremented only on `UPDATE`, this technique allows conflict detection for both `UPDATE` and `DELETE`.

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

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

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

7.4.3 bdr.alter_table_conflict_detection

Allows the table owner to change how conflict detection works for a given table.

Synopsis

```
bdr.alter_table_conflict_detection(relation regclass,  
                                  method text,  
                                  column_name name DEFAULT NULL)
```

Parameters

- **relation** - name of the relation for which to set the new conflict detection method.
- **method** - which conflict detection method to use.
- **column_name** - which column to use for storing of the column detection data; this can be skipped, in which case column name will be automatically chosen based on the conflict detection method. The `row_origin` method does not require extra column for metadata storage.

The recognized methods for conflict detection are:

- **row_origin** - origin of the previous change made on the tuple (see *Origin Conflict Detection* above). This is the only method supported which does not require an extra column in the table.
- **row_version** - row version column (see *Row Version Conflict Detection* above).
- **column_commit_timestamp** - per-column commit timestamps (described in the [CLCD](#) chapter).
- **column_modify_timestamp** - per-column modification timestamp (described in the [CLCD](#) chapter).

Notes

For more information about the difference between `column_commit_timestamp` and `column_modify_timestamp` conflict detection methods, see [Current vs Commit Timestamp](#) section in the [CLCD](#) chapter.

This function uses the same replication mechanism as DDL statements. This means the replication is affected by the [ddl filters](#) configuration.

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

This function is transactional - the effects can be rolled back with the `ROLLBACK` of the transaction, and the changes are visible to the current transaction.

The `bdr.alter_table_conflict_detection` function can be only executed by the owner of the `relation`, unless `bdr.backwards_compatibility` is set to 30618 or below.

!!! Warning Please note that when changing the conflict detection method from one that uses an extra column to store metadata, that column will be dropped.

!!! Warning This function automatically disables CAMO (together with a warning, as long as these are not disabled with `bdr.camo_enable_client_warnings`).

7.4.4 List of Conflict Types

BDR recognizes the following conflict types, which can be used as the `conflict_type` parameter:

- `insert_exists` - an incoming insert conflicts with an existing row via a primary key or an unique key/index.
- `update_differing` - an incoming update's key row differs from a local row. This can only happen when using *Row Version Conflict Detection*.
- `update_origin_change` - an incoming update is modifying a row that was last changed by a different node.
- `update_missing` - an incoming update is trying to modify a row that does not exist.
- `update_recently_deleted` - an incoming update is trying to modify a row that was recently deleted.
- `update_pkey_exists` - an incoming update has modified the PRIMARY KEY to a value that already exists on the node that is applying the change.
- `multiple_unique_conflicts` - the incoming row conflicts with multiple UNIQUE constraints/indexes in the target table.
- `delete_recently_updated` - an incoming delete with an older commit timestamp than the most recent update of the row on the current node, or when using *Row Version Conflict Detection*.
- `delete_missing` - an incoming delete is trying to remove a row that does not exist.
- `target_column_missing` - the target table is missing one or more columns present in the incoming row.
- `source_column_missing` - the incoming row is missing one or more columns that are present in the target table.
- `target_table_missing` - the target table is missing.
- `apply_error_ddl` - an error was thrown by PostgreSQL when applying a replicated DDL command.

7.5 Conflict Resolution

Most conflicts can be resolved automatically. BDR defaults to a last-update-wins mechanism - or more accurately, the `update_if_newer` conflict resolver. This mechanism will retain the most recently inserted or changed row of the two conflicting ones based on the same commit timestamps used for conflict detection. The behaviour in certain corner case scenarios depends on the settings used for `[bdr.create_node_group]` and alternatively for `[bdr.alter_node_group]`.

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

7.5.1 `bdr.alter_node_set_conflict_resolver`

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

Synopsis

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

Parameters

- `node_name` - name of the node that is being changed
- `conflict_type` - conflict type for which the setting should be applied (see [List of Conflict Types](#))
- `conflict_resolver` - which resolver to use for the given conflict type (see [List of Conflict Resolvers](#))

Notes

Currently only the local node can be changed. The function call is not replicated. If you want to change settings on multiple nodes, the function must be run on each of them.

Note that the configuration change made by this function will override any default behaviour of conflict resolutions specified via `[bdr.create_node_group]` or `bdr.alter_node_group`.

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

7.5.2 List of Conflict Resolvers

There are several conflict resolvers available in BDR, with differing coverages of the conflict types they can handle:

- `error` - throws error and stops replication. Can be used for any conflict type.
- `skip` - skips processing of the remote change and continues replication with the next change. Can be used for `insert_exists`, `update_differing`, `update_origin_change`, `update_missing`, `update_recently_deleted`, `update_pkey_exists`, `delete_recently_updated`, `delete_missing`, `target_table_missing`, `target_column_missing` and `source_column_missing` conflict types.
- `skip_if_recently_dropped` - skip the remote change if it's for a table that does not exist on downstream because it has been recently (currently within 1 day) dropped on the downstream; throw an error otherwise. Can be used for the `target_table_missing` conflict type. `skip_if_recently_dropped` conflict resolver may pose challenges if a table with the same name is recreated shortly after it's dropped. In that case, one of the nodes may see the DMLs on the recreated table before it sees the DDL to recreate the table. It will then incorrectly skip the remote data, assuming that the table is recently dropped and cause data loss. It is hence recommended to not reuse the object namesq immediately after they are dropped along with this conflict resolver.
- `skip_transaction` - skips the whole transaction that has generated the conflict. Can be used for `apply_error_ddl` conflict.
- `update_if_newer` - update if the remote row was committed later (as determined by the wall clock of the originating server) than the conflicting local row. If the timestamps are same, the node id is used as a tie-breaker to ensure that same row is picked on all nodes (higher nodeid wins). Can be used for `insert_exists`, `update_differing`, `update_origin_change` and `update_pkey_exists` conflict types.
- `update` - always perform the replicated action. Can be used for `insert_exists` (will turn the INSERT into UPDATE), `update_differing`, `update_origin_change`, `update_pkey_exists`, and `delete_recently_updated` (performs the delete).

- `insert_or_skip` - try to build a new row from available information sent by the origin and INSERT it; if there is not enough information available to build a full row, skip the change. Can be used for `update_missing` and `update_recently_deleted` conflict types.
- `insert_or_error` - try to build new row from available information sent by origin and INSERT it; if there is not enough information available to build full row, throw error and stop the replication. Can be used for `update_missing` and `update_recently_deleted` conflict types.
- `ignore` - ignore any missing target column and continue processing. Can be used for the `target_column_missing` conflict type.
- `ignore_if_null` - ignore a missing target column if the extra column in the remote row contains a NULL value, otherwise throw error and stop replication. Can be used for the `target_column_missing` conflict type.
- `use_default_value` - fill the missing column value with the default (including NULL if that's the column default) and continue processing. Any error while processing the default or violation of constraints (i.e. NULL default on NOT NULL column) will stop replication. Can be used for the `source_column_missing` conflict type.

The `insert_exists`, `update_differing`, `update_origin_change`, `update_missing`, `multiple_unique_conflicts`, `update_recently_deleted`, `update_pkey_exists`, `delete_recently_updated` and `delete_missing` conflict types can also be resolved by user-defined logic using [Conflict Triggers](#).

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

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

7.5.3 Default Conflict Resolvers

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

7.5.4 List of Conflict Resolutions

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

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

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

7.6 Conflict Logging

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

7.6.1 `bdr.alter_node_set_log_config`

Set the conflict logging configuration for a node.

Synopsis

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

Parameters

- `node_name` - name of the node that is being changed
- `log_to_file` - whether to log to the server log file
- `log_to_table` - whether to log to the `bdr.conflict_history` table
- `conflict_type` - which conflict types to log; NULL (the default) means all
- `conflict_resolution` - which conflict resolutions to log; NULL (the default) means all

Notes

Currently only the local node can be changed. The function call is not replicated. If you want to change settings on multiple nodes, the function must be run on each of them.

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

Listing Conflict Logging Configurations

The view `bdr.node_log_config` shows all the logging configurations. It lists the name of the logging configuration, where it logs and which conflict type and resolution it logs.

Logging Conflicts to a Table

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

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

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

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

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

7.6.2 Conflict Reporting

Conflicts logged to tables can be summarized in reports. This allows application owners to identify, understand and resolve conflicts, and/or introduce application changes to prevent them.

```
SELECT nspname, relname
, date_trunc('day', local_time) :: date AS date
, count(*)
FROM bdr.conflict_history
WHERE local_time > date_trunc('day', current_timestamp)
GROUP BY 1,2,3
ORDER BY 1,2;
```

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

7.7 Data Verification with LiveCompare

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

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

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

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

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

Please refer to the LiveCompare docs for further details.

SEQUENCES

Many applications require that unique surrogate ids be assigned to database entries. Often the database `SEQUENCE` object is used to produce these. In PostgreSQL these can be either a manually created sequence using the `CREATE SEQUENCE` command and retrieved by calling `nextval()` function, or `serial` and `bigserial` columns or alternatively `GENERATED BY DEFAULT AS IDENTITY` columns.

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

8.1 BDR Global Sequences

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

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

Using BDR global sequences allows you to avoid the problems with insert conflicts. If you define a `PRIMARY KEY` or `UNIQUE` constraint on a column which is using a global sequence, it is not possible for any node to ever get the same value as any other node. When BDR synchronizes inserts between the nodes, they can never conflict.

BDR global sequences extend PostgreSQL sequences, so are crash-safe. To use them, you must have been granted the `bdr_application` role.

There are various possible algorithms for global sequences:

- Timeshard sequences
- Globally-allocated range sequences

Timeshard sequences generate values using an algorithm that does not require inter-node communication at any point, so is faster and more robust, as well as having the useful property of recording the timestamp at which they were created. Timeshard sequences have the restriction that they work only for 64-bit `BIGINT` datatypes and produce values 19 digits long, which may be too long for use in some host language datatypes such as Javascript Integer types. Globally-allocated sequences allocate a local range of values which can be replenished as-needed by inter-node consensus, making them suitable for either `BIGINT` or `INTEGER` sequences.

A global sequence can be created using the `bdr.alter_sequence_set_kind()` function. This function takes a standard PostgreSQL sequence and marks it as a BDR global sequence. It can also convert the sequence back to the standard PostgreSQL sequence (see below).

BDR also provides the configuration variable `bdr.default_sequence_kind`, which determines what kind of sequence will be created when the `CREATE SEQUENCE` command is executed or when a `serial`, `bigserial` or `GENERATED BY DEFAULT AS IDENTITY` column is created. Valid settings are:

- `local` (the default) meaning that newly created sequences are the standard PostgreSQL (local) sequences.
- `galloc` which always creates globally-allocated range sequences.
- `timeshard` which creates time-sharded global sequences for `BIGINT` sequences, but will throw `ERRORs` when used with `INTEGER` sequences.

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

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

8.1.1 Timeshard Sequences

The ids generated by timeshard sequences are loosely time-ordered so they can be used to get the approximate order of data insertion, like standard PostgreSQL sequences. Values generated within the same millisecond might be out of order, even on one node. The property of loose time-ordering means they are suitable for use as range partition keys.

Timeshard sequences work on one or more nodes, and do not require any inter-node communication after the node join process completes. So they may continue to be used even if there's the risk of extended network partitions, and are not affected by replication lag or inter-node latency.

Timeshard sequences generate unique ids in a different way to standard sequences. The algorithm uses 3 components for a sequence number. The first component of the sequence is a timestamp at the time of sequence number generation. The second component of the sequence number is the unique id assigned to each BDR node, which ensures that the ids from different nodes will always be different. Finally, the third component is the number generated by the local sequence itself.

While adding a unique node id to the sequence number would be enough to ensure there are no conflicts, we also want to keep another useful property of sequences, which is that the ordering of the sequence numbers roughly corresponds to the order in which data was inserted into the table. Putting the timestamp first ensures this.

A few limitations and caveats apply to timeshard sequences.

Timeshard sequences are 64-bits wide and need a `bigint` or `bigserial`. Values generated will be at least 19 digits long. There is no practical 32-bit `integer` version, so cannot be used with `serial` sequences - use globally-allocated range sequences instead.

There is a limit of 8192 sequence values generated per millisecond on any given node for any given sequence. If more than 8192 sequences per millisecond are generated from one sequence on one node, the generated values will wrap around and could collide. There is no check on that for performance reasons; the value is not reset to 0 at the start of each ms. Collision will usually result in a `UNIQUE` constraint violation on `INSERT` or `UPDATE`. It cannot cause a replication conflict, because sequence values generated on different nodes cannot *ever* collide since they contain the `nodeid`.

In practice this is harmless; values are not generated fast enough to trigger this limitation as there will be other work being done, rows inserted, indexes updated, etc. Despite that, applications should have a `UNIQUE` constraint in place where they absolutely rely on a lack of collisions.

Perhaps more importantly, the timestamp component will run out of values in the year 2050, and if used in combination with `bigint`, the values will wrap to negative numbers in the year 2033. This means that sequences generated after 2033 will have negative values. If you plan to deploy your application beyond this date, try one of [UUIDs](#), [KSUUIDs](#) and [Other Approaches](#) mentioned below, or use globally-allocated range sequences instead.

The `INCREMENT` option on a sequence used as input for timeshard sequences is effectively ignored. This could be relevant for applications that do sequence ID caching, like many object-relational mapper (ORM) tools, notably Hibernate. Because the sequence is time-based, this has little practical effect since the sequence will have advanced to a new non-colliding value by the time the application can do anything with the cached values.

Similarly, the `START`, `MINVALUE`, `MAXVALUE` and `CACHE` settings may be changed on the underlying sequence, but there is no benefit to doing so. The sequence's low 14 bits are used and the rest is discarded, so the value range limits do not affect the function's result. For the same reason, `setval()` is not useful for timeshard sequences.

8.1.2 Globally-allocated range Sequences

The globally-allocated range (or `galloc`) sequences allocate ranges (chunks) of values to each node. When the local range is used up, a new range is allocated globally by consensus amongst the other nodes. This uses the key space efficiently, but requires that the local node be connected to a majority of the nodes in the cluster for the sequence generator to progress, when the currently assigned local range has been used up.

Unlike timeshard sequences, `galloc` sequences support all sequence data types provided by PostgreSQL - `smallint`, `integer` and `bigint`. This means that `galloc` sequences can be used in environments where 64-bit sequences are problematic, such as using integers in javascript, since that supports only 53-bit values, or when the sequence is displayed on output with limited space.

The range assigned by each voting is currently predetermined based on the datatype the sequence is using:

- `smallint` - 1 000 numbers
- `integer` - 1 000 000 numbers
- `bigint` - 1 000 000 000 numbers

Each node will allocate two chunks of `seq_chunk_size`, one for the current use plus a reserved chunk for future usage, so the values generated from any one node will increase monotonically. However, viewed globally, the values generated will not be ordered at all. This could cause a loss of performance due to the effects on b-tree indexes, and will typically mean that generated values will not be useful as range partition keys.

The main downside of the `galloc` sequences is that once the assigned range is used up, the sequence generator has to ask for consensus about the next range for the local node that requires inter-node communication, which could lead to delays or operational issues if the majority of the BDR group is not accessible. This may be avoided in later releases.

The `CACHE`, `START`, `MINVALUE` and `MAXVALUE` options work correctly with `galloc` sequences, however you need to set them before transforming the sequence to `galloc` kind. The `INCREMENT BY` option also works correctly, however, you cannot assign an increment value which is equal to or more than the above ranges assigned for each sequence datatype. `setval()` does not reset the global state for `galloc` sequences and should not be used.

A few limitations apply to `galloc` sequences. BDR tracks `galloc` sequences in a special BDR catalog `bdr.sequence_alloc`. This catalog is required to track the currently allocated chunks for the `galloc` sequences. The sequence name and namespace is stored in this catalog. Since the sequence chunk allocation is managed via RAFT whereas any changes to the sequence name/namespace is managed via replication stream, BDR currently does not support renaming `galloc` sequences, or moving them to another namespace or renaming the namespace that contains a `galloc` sequence. The user should be mindful of this limitation while designing application schema.

Usage

Before transforming a local sequence to `galloc`, you need to take care of these prerequisites:

When sequence kind is altered to `galloc`, it will be rewritten and restart from the defined start value of the local sequence. If this happens on an existing sequence in a production database you will need to query the current value then set the start value appropriately. To assist with this use case, BDR allows users to pass a starting value with the function `bdr.alter_sequence_set_kind()`. If you are already using offset and you have writes from multiple nodes, you need to check what is the greatest used value and restart the sequence at least to the next value.

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

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

Since users cannot lock a sequence, you must leave a \$MARGIN value to allow operations to continue while the max() value is queried.

The bdr.sequence_alloc table will give information on the chunk size and what ranges are allocated around the whole cluster. In this example we started our sequence from 333, and we have two nodes in the cluster, we can see that we have a number of allocation 4, that is 2 per node and the chunk size is 1000000 that is related to an integer sequence.

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

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

- Node Node1 is using range from 333 to 2000333.

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

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

- Node Node2 is using range from 2000004 to 4000003.

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

(continues on next page)

(continued from previous page)

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

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

When a node finishes a chunk, it will ask a consensus for a new one and get the first available; in our case, it will be from 4000334 to 5000333. This will be the new reserved chunk and it will start to consume the old reserved chunk.

8.2 UUIDs, KSUUUIDs and Other Approaches

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

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

Please note that BDR applications **cannot** use other methods safely: counter-table based approaches relying on `SELECT ... FOR UPDATE, UPDATE ... RETURNING ...` or similar for sequence generation will not work correctly in BDR, because BDR does not take row locks between nodes. The same values will be generated on more than one node. For the same reason, the usual strategies for “gapless” sequence generation do not work with BDR. In most cases the application should coordinate generation of sequences that must be gapless from some external source using two-phase commit, or it should only generate them on one node in the BDR group.

8.2.1 UUIDs and KSUUUIDs

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

In the incredibly unlikely event of a collision, conflict detection will choose the newer of the two inserted records to retain. Conflict logging, if enabled, will record such an event, but it is *exceptionally* unlikely to ever occur, since collisions only become practically likely after about 2^{64} keys have been generated.

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

BDR provides functions for working with a K-Sortable variant of UUID data, known as KSUUUID, which generates values that can be stored using PostgreSQL's standard UUID data type. A KSUUUID value is similar to UUIDv1 in that it stores both timestamp and random data, following the UUID standard. The difference is that KSUUUID is K-Sortable, meaning that it's weakly sortable by timestamp. This makes it more useful as a database key as it produces more compact btree indexes, which improves the effectiveness of search, and allows natural time-sorting of result data. Unlike UUIDv1, KSUUUID values do not include the MAC of the computer on which they were generated, so there should be no security concerns from using KSUUUIDs.

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

There are two versions of KSUUID in BDR, v1 and v2. The legacy KSUUID v1 is now deprecated but is kept in order to support existing installations and should not be used for new installations. The internal contents of the v1 and v2 are not compatible, and as such the functions to manipulate them are also not compatible. The v2 of KSUUID also no longer stores the UUID version number.

8.2.2 Step & Offset Sequences

In offset-step sequences, a normal PostgreSQL sequence is used on each node. Each sequence increments by the same amount and starts at differing offsets. For example with step 1000, node1's sequence generates 1001, 2001, 3001, and so on, node2's generates 1002, 2002, 3002, etc. This scheme works well even if the nodes cannot communicate for extended periods, but the designer must specify a maximum number of nodes when establishing the schema, and it requires per-node configuration. However, mistakes can easily lead to overlapping sequences.

It is relatively simple to configure this approach with BDR by creating the desired sequence on one node, like this:

```
CREATE TABLE some_table (  
    generated_value bigint primary key  
);  
  
CREATE SEQUENCE some_seq INCREMENT 1000 OWNED BY some_table.generated_value;  
  
ALTER TABLE some_table ALTER COLUMN generated_value SET DEFAULT nextval('some_seq');
```

... then on each node calling `setval()` to give each node a different offset starting value, e.g.:

```
-- On node 1  
SELECT setval('some_seq', 1);  
  
-- On node 2  
SELECT setval('some_seq', 2);  
  
-- ... etc
```

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

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

BDR does not currently offer any automation for configuration of the per-node offsets on such step/offset sequences.

Composite Keys

A variant on step/offset sequences is to use a composite key composed of `PRIMARY KEY (node_number, generated_value)`, where the node number is usually obtained from a function that returns a different number on each node. Such a function may be created by temporarily disabling DDL replication and creating a constant SQL function, or by using a one-row table that is not part of a replication set to store a different value in each node.

8.3 Global Sequence Management Interfaces

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

Note that the following functions are considered to be DDL, so DDL replication and global locking applies to them.

8.3.1 `bdr.alter_sequence_set_kind`

Allows the owner of a sequence to set the kind of a sequence. Once set, `seqkind` is only visible via the `bdr.sequences` view; in all other ways the sequence will appear as a normal sequence.

BDR treats this function as DDL, so DDL replication and global locking applies, if that is currently active. See [DDL Replication].

Synopsis

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

Parameters

- `seqoid` - name or Oid of the sequence to be altered
- `seqkind` - `local` for a standard PostgreSQL sequence, `timeshard` for BDR global sequence which uses the “time and sharding” based algorithm described in the [BDR Global Sequences](#) section, or `gallocc` for globally-allocated range sequences which use consensus between nodes to assign unique ranges of sequence numbers to each node

Notes

When changing the sequence kind to `gallocc`, the first allocated range for that sequence will use the sequence start value as the starting point. When there are already existing values used by the sequence before it was changed to `gallocc`, it is recommended to move the starting point so that the newly generated values will not conflict with the existing ones using the following command:

```
ALTER SEQUENCE seq_name START starting_value RESTART
```

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

The function will take a global DDL lock. It will also lock the sequence locally.

This function is transactional - the effects can be rolled back with the `ROLLBACK` of the transaction, and the changes are visible to the current transaction.

The `bdr.alter_sequence_set_kind` function can be only executed by the owner of the sequence, unless `bdr.backwards_compatibility` is set to 30618 or below.

8.3.2 bdr.extract_timestamp_from_timesthard

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

Synopsis

```
bdr.extract_timestamp_from_timesthard(timesthard_seq bigint)
```

Parameters

- `timesthard_seq` - value of a timesthard sequence

Notes

This function is only executed on the local node.

8.3.3 bdr.extract_nodeid_from_timesthard

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

Synopsis

```
bdr.extract_nodeid_from_timesthard(timesthard_seq bigint)
```

Parameters

- `timesthard_seq` - value of a timesthard sequence

Notes

This function is only executed on the local node.

8.3.4 bdr.extract_localseqid_from_timesthard

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

Synopsis

```
bdr.extract_localseqid_from_timesthard(timesthard_seq bigint)
```

Parameters

- `timeshard_seq` - value of a timeshard sequence

Notes

This function is only executed on the local node.

8.3.5 `bdr.timestamp_to_timeshard`

This function converts a timestamp value to a dummy timeshard sequence value.

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

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

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

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

Synopsis

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

Parameters

- `ts` - timestamp to be used for the timeshard sequence generation

Notes

This function is only executed on local node.

8.4 KSUUID v2 Functions

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

8.4.1 `bdr.gen_ksuuid_v2`

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

The return value is of type “UUID”.

Synopsis

```
bdr.gen_ksuuuid_v2(timestamptz)
```

Notes

This function is only executed on the local node.

8.4.2 bdr.ksuuuid_v2_cmp

This function compares the KSUUUID v2 values.

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

Synopsis

```
bdr.ksuuuid_v2_cmp(uuid, uuid)
```

Parameters

- UUID - KSUUUID v2 to compare

Notes

This function is only executed on local node.

8.4.3 bdr.extract_timestamp_from_ksuuuid_v2

This function extracts the timestamp component of KSUUUID v2. The return value is of type “timestamptz”.

Synopsis

```
bdr.extract_timestamp_from_ksuuuid_v2(uuid)
```

Parameters

- UUID - KSUUUID v2 value to extract timestamp from

Notes

This function is only executed on the local node.

8.5 KSUUID v1 Functions

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

8.5.1 bdr.gen_ksuuid

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

Synopsis

```
bdr.gen_ksuuid()
```

Notes

This function is only executed on the local node.

8.5.2 bdr.uuid_v1_cmp

This function compares the KSUUID v1 values.

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

Synopsis

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

Notes

This function is only executed on the local node.

Parameters

- UUID - KSUUID v1 to compare

8.5.3 `bdr.extract_timestamp_from_ksuuid`

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

Synopsis

<code>bdr.extract_timestamp_from_ksuuid(uuid)</code>
--

Parameters

- UUID - KSUUID v1 value to extract timestamp from

Notes

This function is only executed on the local node.

COLUMN-LEVEL CONFLICT DETECTION

By default, conflicts are resolved at row level. That is, when changes from two nodes conflict, we pick either the local or remote tuple and discard the other one. For example, we may compare commit timestamps for the two conflicting changes and keep the newer one. This ensures that all nodes converge to the same result, and establishes commit-order-like semantics on the whole cluster.

However, in some cases it may be appropriate to resolve conflicts at the column-level rather than the row-level.

Consider a simple example, where we have a table “t” with two integer columns “a” and “b”, and a single row (1, 1). Assume that on one node we execute:

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

... while on another node we concurrently (before receiving the preceding UPDATE) execute:

```
UPDATE t SET b = 100
```

This results in an UPDATE-UPDATE conflict. With the `update_if_newer` conflict resolution, we compare the commit timestamps and keep the new row version. Assuming the second node committed last, we end up with (1, 100), effectively discarding the change to column “a”.

For many use cases this is the desired and expected behaviour, but for some this may be an issue - consider for example a multi-node cluster where each part of the application is connected to a different node, updating a dedicated subset of columns in a shared table. In that case, the different components may step on each other’s toes, overwriting their changes.

For such use cases, it may be more appropriate to resolve conflicts on a given table at the column-level. To achieve that, BDR will track the timestamp of the last change for each column separately, and use that to pick the most recent value (essentially `update_if_newer`).

Applied to the previous example, we’ll end up with (100, 100) on both nodes, despite neither of the nodes ever seeing such a row.

When thinking about column-level conflict resolution, it may be useful to see tables as vertically partitioned, so that each update affects data in only one slice. This eliminates conflicts between changes to different subsets of columns. In fact, vertical partitioning may even be a practical alternative to column-level conflict resolution.

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

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

9.1 Enabling and Disabling Column-Level Conflict Resolution

The Column-Level Conflict Resolution is managed by the `bdr.alter_table_conflict_detection()` function.

9.1.1 Example

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

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

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

db=# SELECT bdr.alter_table_conflict_detection(
db(# 'my_app.test_table'::regclass, 'column_modify_timestamp', 'cts');
alter_table_conflict_detection
-----
t

db=# \d my_app.test_table
```

You will see that the function adds a new `cts` column (as specified in the function call), but it also created two triggers (`BEFORE INSERT` and `BEFORE UPDATE`) that are responsible for maintaining timestamps in the new column before each change.

Also worth mentioning is that the new column specifies `NOT NULL` with a default value, which means that `ALTER TABLE ... ADD COLUMN` does not perform a table rewrite.

Note: We discourage using columns with the `bdr.column_timestamps` data type for other purposes as it may have various negative effects (it switches the table to column-level conflict resolution, which will not work correctly without the triggers etc.).

9.1.2 Listing Table with Column-Level Conflict Resolution

Tables having column-level conflict resolution enabled can be listed with the following query, which detects the presence of a column of type `bdr.column_timestamp`:

```
SELECT nc.nspname, c.relname
FROM pg_attribute a
JOIN (pg_class c JOIN pg_namespace nc ON c.relnamespace = nc.oid)
ON a.attrelid = c.oid
JOIN (pg_type t JOIN pg_namespace nt ON t.typnamespace = nt.oid)
ON a.atttypid = t.oid
WHERE NOT pg_is_other_temp_schema(nc.oid)
AND nt.nspname = 'bdr'
AND t.typname = 'column_timestamps'
AND NOT a.attisdropped
AND c.relkind IN ('r', 'v', 'f', 'p');
```

9.1.3 bdr.column_timestamps_create

This function creates column-level conflict resolution. This is called within `column_timestamp_enable`.

Synopsis

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

Parameters

- `p_source` - The two options are 'current' or 'commit'.
- `p_timestamp` - Timestamp is dependent on the source chosen: if 'commit', then `TIMESTAMP_SOURCE_COMMIT`; if 'current', then `TIMESTAMP_SOURCE_CURRENT`.

9.2 DDL Locking

When enabling or disabling column timestamps on a table, the code uses DDL locking to ensure that there are no pending changes from before the switch, to ensure we only see conflicts with either timestamps in both tuples or neither of them. Otherwise, the code might unexpectedly see timestamps in the local tuple and NULL in the remote one. It also ensures that the changes are resolved the same way (column-level or row-level) on all nodes.

9.3 Current vs Commit Timestamp

An important question is what timestamp to assign to modified columns.

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

It may however have various unexpected effects:

- The timestamp changes during statement execution, so if an `UPDATE` affects multiple rows, each will get a slightly different timestamp. This means that the effects of concurrent changes may get “mixed” in various ways (depending on how exactly the changes performed on different nodes interleave).
- The timestamp is unrelated to the commit timestamp, and using it to resolve conflicts means that the result is not equivalent to the commit order, which means it likely is not serializable.

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

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

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

This can also be disabled using `bdr.column_timestamps_disable`.

Commit timestamps currently have a couple of restrictions that are explained in the “Limitations” section.

9.4 Inspecting Column Timestamps

The column storing timestamps for modified columns is maintained automatically by triggers, and must not be modified directly. It may be useful to inspect the current timestamps value, for example while investigating how a particular conflict was resolved.

There are three functions for this purpose:

- `bdr.column_timestamps_to_text(bdr.column_timestamps)`

This function returns a human-readable representation of the timestamp mapping, and is used when casting the value to `text`:

```
db=# select cts::text from test_table;
               cts
-----
{source: current, default: 2018-09-23 19:24:52.118583+02, map: [2 : 2018-09-23 19:25:02.590677+02]}
(1 row)
```

- `bdr.column_timestamps_to_jsonb(bdr.column_timestamps)`

This function turns a JSONB representation of the timestamps mapping, and is used when casting the value to `jsonb`:

```
db=# select jsonb_pretty(cts::jsonb) from test_table;
      jsonb_pretty
-----
{
  "map": {
    "2": "2018-09-23T19:24:52.118583+02:00"
  },
  "source": "current",
  "default": "2018-09-23T19:24:52.118583+02:00"
}
(1 row)
```

- `bdr.column_timestamps_resolve(bdr.column_timestamps, xid)`

This function updates the mapping with the commit timestamp for the attributes modified by the most recent transaction (if it already committed). This only matters when using the commit timestamp. For example in this case, the last transaction updated the second attribute (with `attnum = 2`):

```
test=# select cts::jsonb from test_table;
               cts
-----
{"map": {"2": "2018-09-23T19:29:55.581823+02:00"}, "source": "commit", "default": "2018-09-23T19:29:55.581823+02:00", "modified": [2]}
(1 row)

db=# select bdr.column_timestamps_resolve(cts, xmin)::jsonb from test_table;
      column_timestamps_resolve
-----
(continues on next page)
```

(continues on next page)

(continued from previous page)

```
{ "map": { "2": "2018-09-23T19:29:55.581823+02:00"}, "source": "commit", "default": "2018-
→09-23T19:29:55.581823+02:00"}
```

(1 row)

9.5 Handling column conflicts using CRDT Data Types

By default, column-level conflict resolution simply picks the value with a higher timestamp and discards the other one. It is however possible to reconcile the conflict in different (more elaborate) ways, for example using CRDT types that allow “merging” the conflicting values without discarding any information.

While pglogical does not include any such data types, it allows adding them separately and registering them in a catalog `crdt_handlers`. Aside from the usual data type functions (input/output, ...) each CRDT type has to implement a merge function, which takes exactly three arguments (local value, old remote value, new remote value) and produces a value merging information from those three values.

9.6 Limitations

- The attributes modified by an `UPDATE` are determined by comparing the old and new row in a trigger. This means that if the attribute does not change a value, it will not be detected as modified even if it is explicitly set. For example, `UPDATE t SET a = a` will not mark `a` as modified for any row. Similarly, `UPDATE t SET a = 1` will not mark `a` as modified for rows that are already set to 1.
- For `INSERT` statements, we do not have any old row to compare the new one to, so we consider all attributes to be modified and assign them a new timestamp. This applies even for columns that were not included in the `INSERT` statement and received default values. We could detect which attributes have a default value, but it is not possible to decide if it was included automatically or specified explicitly by the user.

This effectively means column-level conflict resolution does not work for `INSERT-INSERT` conflicts (even if the `INSERT` statements specify different subsets of columns, because the newer row will have all timestamps newer than the older one).

- By treating the columns independently, it is easy to violate constraints in a way that would not be possible when all changes happen on the same node. Consider for example a table like this:

```
CREATE TABLE t (id INT PRIMARY KEY, a INT, b INT, CHECK (a > b));
INSERT INTO t VALUES (1, 1000, 1);
```

...and assume one node does:

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

...while another node does concurrently:

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

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

- The column storing timestamp mapping is managed automatically. Do not specify or override the value in your queries, as it may result in unpredictable effects (we do ignore the value where possible anyway).

- The timestamp mapping is maintained by triggers, but the order in which triggers execute does matter. So if you have custom triggers that modify tuples and are executed after the `pgl_cld_` triggers, the modified columns will not be detected correctly.
- When using regular timestamps to order changes/commits, it is possible that the conflicting changes have exactly the same timestamp (because two or more nodes happened to generate the same timestamp). This risk is not unique to column-level conflict resolution, as it may happen even for regular row-level conflict resolution, and we use node id as a tie-breaker in this situation (the higher node id wins), which ensures that same changes are applied on all nodes.
- It is possible that there is a clock skew between different nodes. While it may induce somewhat unexpected behavior (discarding seemingly newer changes because the timestamps are inverted), clock skew between nodes can be managed using the parameters `bdr.maximum_clock_skew` and `bdr.maximum_clock_skew_action`.
- The underlying pglogical subscription must not discard any changes, which could easily cause divergent errors (particularly for CRDT types). The subscriptions must have `ignore_redundant_updates` set to `false` (which is the default).

Existing groups created with non-default value for `ignore_redundant_updates` can be altered like this:

```
SELECT bdr.alter_node_group_config('group', ignore_redundant_updates := false);
```

CONFLICT-FREE REPLICATED DATA TYPES

Conflict-free replicated data types (CRDT) support merging values from concurrently modified rows, instead of discarding one of the rows (which is what traditional conflict resolution does).

Each CRDT type is implemented as a separate PostgreSQL data type, with an extra callback added to the `bdr.crdt_handlers` catalog. The merge process happens within `pglogical` on the apply side; no additional user action is required.

CRDTs require the table to have column-level conflict resolution enabled as documented in the [CLCD](#) chapter.

The only action taken by the user is the use of a particular data type in `CREATE/ALTER TABLE`, rather than standard built-in data types such as `integer`; e.g. consider the following table with one regular integer counter and a single row:

```
CREATE TABLE non_crdt_example (  
    id integer PRIMARY KEY,  
    counter integer NOT NULL DEFAULT 0  
);  
  
INSERT INTO non_crdt_example (id) VALUES (1);
```

If we issue the following SQL on two nodes at same time:

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

... the resulting values can be seen using this query, after both updates are applied:

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

...showing that we've lost one of the increments, due to the `update_if_newer` conflict resolver. If you use the CRDT counter data type instead, you should observe something like this:

```
CREATE TABLE crdt_example (  
    id integer PRIMARY KEY,  
    counter bdr.crdt_gcounter NOT NULL DEFAULT 0  
);  
  
ALTER TABLE crdt_example REPLICA IDENTITY FULL;
```

(continues on next page)

(continued from previous page)

```
SELECT bdr.alter_table_conflict_detection('crdt_example',
    'column_modify_timestamp', 'cts');

INSERT INTO crdt_example (id) VALUES (1);
```

Again we issue the following SQL on two nodes at same time, then wait for the changes to be applied:

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

SELECT id, counter FROM crdt_example WHERE id = 1;
 id|  counter
----+-----
  1|        2
(1 row)
```

This shows that CRDTs correctly allow accumulator columns to work, even in the face of asynchronous concurrent updates that otherwise conflict.

The `crdt_gcounter` type is an example of state-based CRDT types, that work only with reflexive UPDATE SQL, such as `x = x + 1`, as shown above.

The `bdr.crdt_raw_value` configuration option determines whether queries return the current value or the full internal state of the CRDT type. By default only the current numeric value is returned. When set to `true`, queries return representation of the full state - the special hash operator (`#`) may be used to request only the current numeric value without using the special operator (this is the default behavior). If the full state is dumped using `bdr.crdt_raw_value = on` then the value would only be able to be reloaded with `bdr.crdt_raw_value = on`.

Note: The `bdr.crdt_raw_value` applies only formatting of data returned to clients; i.e. simple column references in the select list. Any column references in other parts of the query (e.g. WHERE clause or even expressions in the select list) may still require use of the `#` operator.

Another class of CRDT data types exists, which we refer to as “delta CRDT” types (and are a special subclass of operation-based CRDTs, as explained later).

With delta CRDTs, any update to a value is automatically compared to the previous value on the same node and then a change is applied as a delta on all other nodes.

```
CREATE TABLE crdt_delta_example (
    id          integer          PRIMARY KEY,
    counter     bdr.crdt_delta_counter NOT NULL DEFAULT 0
);

ALTER TABLE crdt_delta_example REPLICA IDENTITY FULL;

SELECT bdr.alter_table_conflict_detection('crdt_delta_example',
    'column_modify_timestamp', 'cts');

INSERT INTO crdt_delta_example (id) VALUES (1);
```

If we issue the following SQL on two nodes at same time:

```
UPDATE crdt_delta_example
    SET counter = 2    -- notice NOT counter = counter + 2
```

(continues on next page)

(continued from previous page)

```
WHERE id = 1;
```

The resulting values can be seen using this query, after both updates are applied:

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

With a regular integer column the result would be 2, of course. But when we UPDATE the row with a delta CRDT counter, we start with the OLD row version, make a NEW row version and send both to the remote node, where we compare them with the version we find there (let's call that the LOCAL version). Standard CRDTs merge the NEW and the LOCAL version, while delta CRDTs compare the OLD and NEW versions and apply the delta to the LOCAL version.

The CRDT types are installed as part of bdr into the bdr schema. For convenience, the basic operators (+, # and !) and a number of common aggregate functions (min, max, sum and avg) are created in pg_catalog, to make them available without having to tweak search_path.

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

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

10.1 State-based and operation-based CRDTs

Following the notation from [1], we do implement both operation-based and state-based CRDTs.

10.1.1 Operation-based CRDT Types (CmCRDT)

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

Currently, we implement these operation-based CRDTs:

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

These types leverage existing data types (e.g. `crdt_delta_counter` is a domain on a `bigint`), with a little bit of code to compute the delta.

This approach is possible only for types where we know how to compute the delta, but the result is very simple and cheap (both in terms of space and CPU), and has a couple of additional benefits (e.g. we can leverage operators / syntax for the under-lying data type).

The main disadvantage is that it's not possible to reset this value reliably in an asynchronous and concurrent environment.

Note: We could also implement more complicated operation-based types by creating custom data types, storing the state and the last operation (we decode and transfer every individual change, so we don't need multiple operations). But at that point we lose the main benefits (simplicity, reuse of existing data types) without gaining any advantage compared to state-based types (still no capability to reset, ...), except for the space requirements (we don't need a per-node state).

10.1.2 State-based CRDT Types (CvCRDT)

State-based types require a more complex internal state, and so can't use the regular data types directly the way operation-based types do.

Currently, we implement four state-based CRDTs:

- `crdt_gcounter` - `bigint` counter (increment-only)
- `crdt_gsum` - `numeric` sum/counter (increment-only)
- `crdt_pncounter` - `bigint` counter (increments/decrements)
- `crdt_pnsum` - `numeric` sum/counter (increments/decrements)

The internal state typically includes per-node information, increasing the on-disk size but allowing additional benefits. The need to implement custom data types implies more code (in/out functions and operators).

The advantage is the ability to reliably reset the values, a somewhat self-healing nature in the presence of lost changes (which should not happen in properly-operated cluster), and the ability to receive changes from other than source nodes.

Consider for example that a value is modified on node A, and the change gets replicated to B, but not C (due to network issue between A and C). If B modifies the value, and this change gets replicated to C, it will include even the original change from A. With operation-based CRDTs the node C would not receive the change until the A-C network connection starts working again.

The main disadvantages of CvCRDTs are higher costs, both in terms of disk space - we need a bit of information for each node, including nodes that have been already removed from the cluster). The complex nature of the state (serialized into varlena types) means increased CPU usage.

10.2 Disk-Space Requirements

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

For operation-based types this is rather trivial, because the types are merely domains on top of other types, and so have the same disk space requirements (no matter how many nodes are there).

- `crdt_delta_counter` - same as `bigint` (8 bytes)
- `crdt_delta_sum` - same as `numeric` (variable, depending on precision and scale)

There is no dependency on the number of nodes, because operation-based CRDT types do not store any per-node information.

For state-based types the situation is more complicated. All the types are variable-length (stored essentially as a `bytea` column), and consist of a header and a certain amount of per-node information for each node that *modified* the value.

For the `bigint` variants, formulas computing approximate size are (*N* denotes the number of nodes that modified this value):

- `crdt_gcounter` - 32B (header) + *N* * 12B (per-node)
- `crdt_pncounter` - 48B (header) + *N* * 20B (per-node)

For the `numeric` variants there is no exact formula, because both the header and per-node parts include `numeric` variable-length values. To give you an idea of how many such values we need to keep:

- `crdt_gsum`
 - fixed: 20B (header) + *N* * 4B (per-node)
 - variable: (2 + *N*) `numeric` values

- `crdt_pnsum`
 - fixed: 20B (header) + N * 4B (per-node)
 - variable: (4 + 2 * N) numeric values

Note: It does not matter how many nodes are in the cluster, if the values are never updated on multiple nodes. It also does not matter if the updates were concurrent (causing a conflict) or not.

Note: It also does not matter how many of those nodes were already removed from the cluster. There is no way to compact the state yet.

10.3 CRDT Types vs Conflicts Handling

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

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

Note: This makes CRDT types somewhat more expensive compared to plain conflict resolution, because the merge needs to happen every time, even when the conflict resolution can use one of the fast-paths (modified in the current transaction, etc.).

10.4 CRDT Types vs. Conflict Reporting

By default, detected conflicts are written into the server log. Without CRDT types this makes perfect sense, because the conflict resolution essentially throws away one half of the available information (local or remote row, depending on configuration). This presents a data loss.

CRDT types allow both parts of the information to be combined without throwing anything away, eliminating the data loss issue. This makes the conflict reporting unnecessary.

For this reason, we skip the conflict reporting when the conflict can be fully-resolved by CRDT merge, that is if each column meets at least one of these two conditions:

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

Note: This means we also skip the conflict reporting when there are no CRDT columns, but all values in local/remote tuples are equal.

10.5 Resetting CRDT Values

Resetting CRDT values is possible but requires special handling. The asynchronous nature of the cluster means that different nodes may see the reset operation (no matter how it's implemented) at different places in the change stream. Different nodes may also initiate a reset concurrently; i.e. before observing the reset from the other node.

In other words, to make the reset operation behave correctly, it needs to be commutative with respect to the regular operations. Many naive ways to reset a value (which may work perfectly well on a single-node) fail for exactly this reason.

For example, the simplest approach to resetting a value might be:

```
UPDATE crdt_table SET cnt = 0 WHERE id = 1;
```

With state-based CRDTs this does not work - it throws away the state for the other nodes, but only locally. It will be added back by merge functions on remote nodes, causing diverging values, and eventually receiving it back due to changes on the other nodes.

With operation-based CRDTs, this may seem to work because the update is interpreted as a subtraction of `-cnt`. But it only works in the absence of concurrent resets. Once two nodes attempt to do a reset at the same time, we'll end up applying the delta twice, getting a negative value (which is not what we expected from a reset).

It might also seem that `DELETE + INSERT` can be used as a reset, but this has a couple of weaknesses too. If the row is reinserted with the same key, it's not guaranteed that all nodes will see it at the same position in the stream of operations (with respect to changes from other nodes). BDR specifically discourages re-using the same Primary Key value since it can lead to data anomalies in concurrent cases.

State-based CRDT types can reliably handle resets, using a special `!` operator like this:

```
UPDATE tab SET counter = !counter WHERE ...;
```

By “reliably” we mean the values do not have the two issues illustrated above - multiple concurrent resets and divergence.

Operation-based CRDT types can only be reset reliably using [Eager Replication](#), since this avoids multiple concurrent resets. Eager Replication can also be used to set either kind of CRDT to a specific value.

10.6 Implemented CRDT data types

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

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

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

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

10.6.1 grow-only counter (`crdt_gcounter`)

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

```

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

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

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

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

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

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

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

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

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

```

10.6.2 grow-only sum (crdt_gsum)

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

```

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

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

```

(continues on next page)

(continued from previous page)

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

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

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

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

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

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

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

10.6.3 positive-negative counter (crdt_pncounter)

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

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

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

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

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

(continues on next page)

(continued from previous page)

```

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

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

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

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

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

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

```

10.6.4 positive-negative sum (crdt_pnsum)

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

```

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

INSERT INTO crdt_test VALUES (1, 0);           -- initialized to 0
INSERT INTO crdt_test VALUES (2, 1298.24);    -- initialized to 1298.24
INSERT INTO crdt_test VALUES (3, -45.31);     -- initialized to -45.31

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

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

-- decrement sums
UPDATE crdt_test SET pnsum = pnsum - 7.3       WHERE id = 1;

```

(continues on next page)

(continued from previous page)

```

UPDATE crdt_test SET pnsun = pnsun - 192.83 WHERE id = 2;
UPDATE crdt_test SET pnsun = pnsun - (-12.22) WHERE id = 3;

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

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

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

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

```

10.6.5 delta counter (crdt_delta_counter)

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

```

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

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

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

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

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

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

```

10.6.6 delta sum (crdt_delta_sum)

- is defined as a numeric domain, so works exactly like a numeric column
- supports increments with both positive and negative values
- is compatible with simple assignments like `sum = value` (common when the new value is computed somewhere in the application)
- no simple way to reset the value (reliably)

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

INSERT INTO crdt_test VALUES (1, 0);           -- initialized to 0
INSERT INTO crdt_test VALUES (2, 129.824);    -- initialized to 129824
INSERT INTO crdt_test VALUES (3, -4.531);    -- initialized to -4531

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

-- increment counters
UPDATE crdt_test SET dsum = dsum + 1.32 WHERE id = 1;
UPDATE crdt_test SET dsum = dsum + 12.01 WHERE id = 2;
UPDATE crdt_test SET dsum = dsum + (-2.4) WHERE id = 3;

-- decrement counters
UPDATE crdt_test SET dsum = dsum - 7.33 WHERE id = 1;
UPDATE crdt_test SET dsum = dsum - 19.83 WHERE id = 2;
UPDATE crdt_test SET dsum = dsum - (-1.2) WHERE id = 3;

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

[1] https://en.wikipedia.org/wiki/Conflict-free_replicated_data_type

DURABILITY & PERFORMANCE OPTIONS

Durability & Performance Options

EAGER REPLICATION

Eager Replication

COMMIT AT MOST ONCE

Commit at Most Once([https://\(documentation.2ndquadrant.com/bdr3-enterprise/release/latest/camo/\)](https://(documentation.2ndquadrant.com/bdr3-enterprise/release/latest/camo/)))

DATABASE SCALING

AutoPartition

TIMESTAMP-BASED SNAPSHOTS

Timestamp-Based Snapshots

REPLICATION SETS

Replication Sets

STREAM TRIGGERS

Stream Triggers

BACKUP AND RECOVERY

Backup and Recovery

UPGRADES

Upgrades

EXPLICIT TWO-PHASE COMMIT (2PC)

Two-Phase Commit

CATALOGS AND VIEWS

Catalogs and Views

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.

22.1 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.

22.2 Monitoring Node Join and Removal

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

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

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

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

Also, the table `bdr.node_catchup_info <catalogs>`__` 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
```

22.3 Monitoring Replication Peers

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

- `bdr.node_slots` <catalogs>`\_\_` for monitoring outgoing replication
- `bdr.subscription_summary` <catalogs>`\_\_` 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` <<https://www.postgresql.org/docs/current/static/monitoring-stats.html#PG-STAT-REPLICATION-VIEW>>`\_\_` and `pg_catalog.pg_replication_slots` <<https://www.postgresql.org/docs/current/view-pg-replication-slots.html>>`\_\_`.

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` <nodes>`\_\_`.

22.3.1 Monitoring Outgoing Replication

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

- `bdr.node_replication_rates` <catalogs>`\_\_` 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.

22.3.2 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
```

22.3.3 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| 0000000000000000000000000000000014000000000000000
 2059916| t| bdr_postgres_bdrgroup_decoder| 0000000000000000000000000000000014000000000000000
(3 rows)
```

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

22.4 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.

22.5 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 `<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.

22.6 Monitoring Conflicts

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

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

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

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

22.7 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.

22.7.1 bdr.set_node_location

This function inserts node metadata into `bdr.node_location`

Synopsis

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

Parameters

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

22.7.2 bdr.set_network_path_info

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

Synopsis

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

Parameters

- `node_group_name` - name of the BDR group
- `region1` - the origin server name
- `region2` - the remote server name
- `location1` - the origin 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

22.7.3 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.

22.8 Apply Statistics

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

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

- Number of INSERTs/UPDATEs/DELETEs/TRUNCATEs replicated
- Block accesses and cache hit ratio
- Total I/O time for read/write

and for relations only, these statistics:

- Total time spent processing replication for the relation

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

and for subscriptions only, these statistics:

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

Tracking of these statistics is controlled by the pglogical GUCs `pglogical.track_subscription_apply` and `pglogical.track_relation_apply` respectively - for details, see [pglogical Settings for BDR](configuration.md#pglogical Settings for BDR).

The example output from these would look like this:

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

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

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

22.9 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 UPDATES and DELETES 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](#).

22.10 Monitoring BDR Versions

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

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

```
bdrdb=# SELECT node_name, postgres_version, pglogical_version,
bdr_version FROM bdr.group_versions_details;
 node_name| postgres_version| pglogical_version| bdr_version
-----+-----+-----+-----
 node1| 13.3| 3.7.10| 3.7.10
 node2| 13.3| 3.7.10| 3.7.10
```

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

For monitoring purposes, we recommend the following alert levels:

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

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

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

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

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

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

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

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

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

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

```
bdrdb=# SELECT * FROM bdr.monitor_group_raft();
 status |          message
-----+-----
 OK | Raft Consensus is working correctly
```

22.12 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, on the other hand, is inactive most of the time. BDR keeps this slot and advances LSN, as all other peers have already consumed the corresponding transactions. So it is not possible to monitor the status (active or inactive) of the group slot.

We recommend the following monitoring alert levels:

- status=UNKNOWN, message=This node is not part of any BDR group
- status=OK, message=All BDR replication slots are working correctly
- status=CRITICAL, message=There is at least 1 BDR replication slot which is inactive
- status=CRITICAL, message=There is at least 1 BDR replication slot which is missing

The described behavior is implemented in the function `bdr.monitor_local_replslots()`, which uses replication slot status information returned from view `bdr.node_slots` (slot active or inactive) to provide a local check considering all BDR node replication slots, except the BDR group slot.

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

(continues on next page)

(continued from previous page)

```
OK| All BDR replication slots are working correctly
```

22.13 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`.

BDR SYSTEM FUNCTIONS

BDR management is primarily accomplished via SQL-callable functions. All functions in BDR are exposed in the `bdr` schema. Any calls to these functions should be schema-qualified, rather than putting `bdr` in the `search_path`.

This page contains additional system functions that are not described in the other sections of the documentation.

Note that you cannot manipulate BDR-owned objects using pglogical functions; only using the following supplied functions.

23.1 Version Information Functions

23.1.1 `bdr.bdr_edition`

This function returns a textual representation of the BDR edition. BDR3 is distributed in either Standard Edition (SE) or Enterprise Edition (EE); this function can be used to check which of those is currently installed. Deprecated.

23.1.2 `bdr.bdr_version`

This function retrieves the textual representation of the BDR version that is currently in use.

23.1.3 `bdr.bdr_version_num`

This function retrieves a numerical representation of the BDR version that is currently in use. Version numbers are monotonically increasing, allowing this value to be used for less-than and greater-than comparisons.

The following formula is used to turn the version number consisting of major version, minor version and patch release into a single numerical value:

$\text{MAJOR_VERSION} * 100000 + \text{MINOR_VERSION} * 100 + \text{PATCH_RELEASE}$
--

23.1.4 bdr.wal_sender_stats

If the [Decoding Worker](#) is enabled, this view shows information about the decoder slot and current LCR (Logical Change Record) segment file being read by each WAL sender.

Synopsis

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

Output columns

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

23.2 System and Progress Information Parameters

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

23.2.1 bdr.local_node_id

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

It is also used in combination with CAMO, see the [Connection pools and proxies](#) section.

23.2.2 bdr.last_committed_lsn

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

23.2.3 transaction_id

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

!!! Note This is only available on EDB Postgres Extended.

23.3 Utility Functions

23.3.1 `bdr.wait_slot_confirm_lsn`

Allows the user to wait until the last write on this session has been replayed to one or all nodes.

Waits until a slot passes certain LSN. If no position is supplied, the current write position is used on the local node.

If no slot name is passed, it will wait until all BDR slots pass the LSN. This is a separate function from the one provided by `pglogical` so that we can only wait for slots registered for other BDR nodes, not all `pglogical` slots and, more importantly, not our BDR group slot.

The function polls every 1000ms for changes from other nodes.

If a slot is dropped concurrently the wait will end for that slot. If a node is currently down and is not updating its slot then the wait will continue. You may wish to set `statement_timeout` to complete earlier in that case.

Synopsis

```
bdr.wait_slot_confirm_lsn(slot_name text DEFAULT NULL, target_lsn pg_lsn DEFAULT NULL)
```

Parameters

- `slot_name` - name of replication slot, or if NULL, all BDR slots (only)
- `target_lsn` - LSN to wait for, or if NULL, use the current write LSN on the local node

23.3.2 `bdr.wait_for_apply_queue`

The function `bdr.wait_for_apply_queue` allows a BDR node to wait for the local application of certain transactions originating from a given BDR node. It will return only after all transactions from that peer node are applied locally. An application or a proxy can use this function to prevent stale reads.

For convenience, BDR provides a special variant of this function for CAMO and the CAMO partner node, see `bdr.wait_for_camo_partner_queue`.

In case a specific LSN is given, that's the point in the recovery stream from the peer to wait for. This can be used in combination with `bdr.last_committed_lsn` retrieved from that peer node on a previous or concurrent connection.

If the given `target_lsn` is NULL, this function checks the local receive buffer and uses the LSN of the last transaction received from the given peer node. Effectively waiting for all transactions already received to be applied. This is especially useful in case the peer node has failed and it's not known which transactions have been sent. Note that in this case, transactions that are still in transit or buffered on the sender side are not waited for.

Synopsis

```
bdr.wait_for_apply_queue(peer_node_name TEXT, target_lsn pg_lsn)
```

Parameters

- `peer_node_name` - the name of the peer node from which incoming transactions are expected to be queued and which should be waited for. If NULL, waits for all peer node's apply queue to be consumed.
- `target_lsn` - the LSN in the replication stream from the peer node to wait for, usually learned via `bdr.last_committed_lsn` from the peer node.

23.3.3 bdr.get_node_sub_receive_lsn

This function can be used on a subscriber to get the last LSN that has been received from the given origin. Either filtered to take into account only relevant LSN increments for transactions to be applied or unfiltered.

The difference between the output of this function and the output of `bdr.get_node_sub_apply_lsn()` measures the size of the corresponding apply queue.

Synopsis

```
bdr.get_node_sub_receive_lsn(node_name name, committed bool default true)
```

Parameters

- `node_name` - the name of the node which is the source of the replication stream whose LSN we are retrieving/
- `committed` - the default (true) makes this function take into account only commits of transactions received, rather than the last LSN overall; including actions that have no effect on the subscriber node.

23.3.4 bdr.get_node_sub_apply_lsn

This function can be used on a subscriber to get the last LSN that has been received and applied from the given origin.

Synopsis

```
bdr.get_node_sub_apply_lsn(node_name name)
```

Parameters

- `node_name` - the name of the node which is the source of the replication stream whose LSN we are retrieving.

23.3.5 bdr.run_on_all_nodes

Function to run a query on all nodes.

!!! Warning This function will run an arbitrary query on a remote node with the privileges of the user used for the internode connections as specified in the node's DSN. Caution needs to be taken when granting privileges to this function.

Synopsis

```
bdr.run_on_all_nodes(query text)
```

Parameters

- `query` - arbitrary query to be executed.

Notes

This function will connect to other nodes and execute the query, returning a result from each of them in json format. Multiple rows may be returned from each node, encoded as a json array. Any errors, such as being unable to connect because a node is down, will be shown in the response field. No explicit `statement_timeout` or other runtime parameters are set, so defaults will be used.

This function does not go through normal replication, it uses direct client connection to all known nodes. By default, the connection is created with `bdr.ddl_replication = off`, since the command are already being sent to all of the nodes in the cluster.

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

Example

It's useful to use this function in monitoring, for example in the following query:

```
SELECT bdr.run_on_all_nodes($$
    SELECT local_slot_name, origin_name, target_name, replay_lag_size
    FROM bdr.node_slots
    WHERE origin_name IS NOT NULL
$$);
```

... will return something like this on a two node cluster:

```
[
  {
    "dsn": "host=node1 port=5432 dbname=bdrdb user=postgres ",
    "node_id": "2232128708",
```

(continues on next page)

(continued from previous page)

```

    "response": {
      "command_status": "SELECT 1",
      "command_tuples": [
        {
          "origin_name": "node1",
          "target_name": "node2",
          "local_slot_name": "bdr_bdrdb_bdrgroup_node2",
          "replay_lag_size": "0 bytes"
        }
      ]
    },
    "node_name": "node1"
  },
  {
    "dsn": "host=node2 port=5432 dbname=bdrdb user=postgres ",
    "node_id": "2058684375",
    "response": {
      "command_status": "SELECT 1",
      "command_tuples": [
        {
          "origin_name": "node2",
          "target_name": "node1",
          "local_slot_name": "bdr_bdrdb_bdrgroup_node1",
          "replay_lag_size": "0 bytes"
        }
      ]
    },
    "node_name": "node2"
  }
]

```

23.3.6 bdr.global_lock_table

This function will acquire a global DML locks on a given table. See [DDL Locking Details](#) for information about global DML lock.

Synopsis

```
bdr.global_lock_table(relation regclass)
```

Parameters

- `relation` - name or Oid of the relation to be locked.

Notes

This function will acquire the global DML lock independently of the `ddl_locking` setting.

The `bdr.global_lock_table` function requires UPDATE, DELETE, or TRUNCATE privilege on the locked `relation`, unless `bdr.backwards_compatibility` is set to 30618 or below.

23.3.7 bdr.wait_for_xid_progress

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

Synopsis

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

Parameters

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

Notes

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

23.3.8 bdr.local_group_slot_name

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

Example

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

23.3.9 bdr.node_group_type

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

Example

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

23.4 Global Advisory Locks

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

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

Advisory lock is transactional in nature. So the lock is automatically released when the transaction ends unless it is explicitly released before the end of the transaction, in which case it will be available as soon as it's released. Session level advisory locks are not currently supported.

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

23.4.1 bdr.global_advisory_lock

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

Synopsis

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

parameters

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

Synopsis

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

parameters

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

23.4.2 bdr.global_advisory_unlock

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

Synopsis

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

parameters

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

Synopsis

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

parameters

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

CREDITS AND LICENCE

BDR3 has been designed, developed and tested by this team:

- Petr Jelinek
- Craig Ringer
- Markus Wanner
- Pavan Deolasee
- Tomas Vondra
- Simon Riggs
- Nikhil Sontakke
- Pallavi Sontakke
- Amruta Deolasee
- Rahila Syed
- Ashutosh Bapat
- Abhijit Menon-Sen
- Florin Irion
- Oliver Riggs

Copyright © 2018-2020 2ndQuadrant Ltd Copyright © 2021 EnterpriseDB UK Ltd

BDR3 is provided under EDB usage licenses.

The reproduction of these documents is prohibited.

25.1 BDR 3.7.12

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

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

25.1.1 Improvements

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

25.1.2 Resolved Issues

- Fix a potential data loss issue with `bdr_init_physical` (RT71888)
When reusing a slot name, previous state was not properly cleaned up in all cases. This has caused potential data loss during physical join as the slot is created ahead of time by `bdr_init_physical` with the same name. The transition from physical to logical replication could miss part of the replication stream, as this drops and recreates the slot. This release properly cleans slot information when dropped and thereby prevents data loss.
- Fix `bdr.camo_local_mode_delay` to really kick in (BDR-1352)

This artificial delay allows throttling a CAMO node that is not currently connected to its CAMO partner to prevent it from producing transactions faster than the CAMO partner can possibly apply. In previous versions, it did not properly kick in after `bdr.global_commit_timeout` amount of lag, but only 1000 times later (due to erroneously comparing seconds to milliseconds).

- Prevent segfault in combination with third-party output plugins (BDR-1424, RT72006)
Adjust handling of logical WAL messages specific to BDR's Eager All Node Replication mode for output plugins unrelated to BDR. This allows for example Debezium's decoderbufs output plugin to work alongside BDR.
- Improve compatibility with Postgres 13 (BDR-1396)
Adjust to an API change in `ReplicationSlotAcquire` that may have led to unintended blocking when non-blocking was requested and vice versa. This version of PGLogical eliminates this potential problem, which has not been observed on production systems so far.
- Fix serialization of Raft snapshots including commit decisions (CAMO, BDR-1454)
A possible mismatch in number of tuples could lead to serialization or deserialization errors for a Raft snapshot taken after transactions using CAMO or Eager All Node replication were used recently and stored their commit decisions.
- Fix `--recovery-conf` option in `bdr_init_physical`

25.1.3 Upgrades

This release supports upgrading from following versions of BDR:

- 3.7.9 and higher
- 3.6.27

25.2 BDR 3.7.11

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

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

25.2.1 Improvements

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

25.2.2 Resolved Issues

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

25.2.3 Upgrades

This release supports upgrading from following versions of BDR:

- 3.7.9 and higher
- 3.6.27

25.3 BDR 3.7.10

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

25.3.1 Improvements

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

25.3.2 Resolved Issues

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

25.4 BDR 3.7.9

25.4.1 Improvements

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

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

25.4.2 Resolved Issues

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

- Fix handling of `skip_transaction` conflict resolver when there are multiple changes in the transaction after the one that caused the `skip_transaction` (BDR-886)
- Fix Raft snapshot creation for autopartitioned tables (RT71178, BDR-955)
Previously the Raft snapshot didn't take into account state of autopartition tasks on all nodes when writing the information. This could result in some nodes skipping partition creation after prolonged period of downtime.
- Adjust transaction and snapshot handling in autopartition (BDR-903)
This ensures valid snapshot is used during autopartition processing at all times. The previous approach would cause problem in the future point release of PostgreSQL.
- Fix KSUID column detection in autopartition
- Fix misreporting of node status by `bdr.drop_node()` function
- Ensure that correct sequence type is always set in the global gallo sequence state.
- Fix DDL replication and locking management of several commands (BDR-874)
`ANALYZE`, `CHECKPOINT`, `CLUSTER`, `PREPARE/COMMIT/ABORT TRANSACTION`, `MOVE`, `RELEASE`, `ROLLBACK` were documented as replicated and some of these even tried to take DDL lock which they should not.
- Reduce logging of some unreplicated utility commands (BDR-874)
`PREPARE` and `EXECUTE` don't need to spam about not being replicated as nobody expects that they would be.
- Fix global locking of `ALTER TABLE ... SET` (BDR-653)
It should not take global DML lock.
- Fix documentation about how `TRUNCATE` command is replicated (BDR-874)
While `TRUNCATE` can acquire global locks, it's not replicated the way other DDL commands are, it's replicated like DML, according to replication set settings.
- Document that CAMO and Eager currently don't work with Decoding Worker (BDR-584)
- Multiple typo and grammar fixes in docs.

25.5 BDR 3.7.8

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

25.5.1 Important Notes

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

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

25.5.2 The Highlights of BDR 3.7

- Support for PostgreSQL 11, 12 and 13
- Support EDB Advanced Server
Both Standard Edition and Enterprise Edition are now available to use with EDB Advanced Server
- Parallel Apply
Allows configuring number of parallel writers that apply the replication stream. This feature is supported in Enterprise Edition only.
- AutoPartition
Allows automatic management of partitioned tables, with automated creation, automated cleanup with configurable retention periods and more.
- Introduce option to separate BDR WAL decoding worker
This allows using single decoding process on each node, regardless of number of subscriptions connected to it. The decoded information is stored in logical change record (LCR) files which are streamed to the other nodes in similar way traditional WAL is. Optional separation of decoding from walsender.
This feature is supported in Enterprise Edition only.
- Implement the concept of subscriber-only nodes
These are wholly joined nodes, but they don't ever send replication changes to other BDR nodes in the cluster. But they do receive changes from all nodes in the cluster (except, of course the other subscriber-only nodes). They do not participate in the RAFT voting protocol, and hence their presence (or absence) does not determine RAFT leader election. We don't need to create any replication slots on these nodes since they don't send replication changes. Similarly, we don't need to create any subscriptions for these nodes on other BDR nodes.
- Support CREATE TABLE ... AS and SELECT INTO statement
This feature is now supported in Enterprise Edition only.
- New ability to define BDR sub-groups in order to better represent physical configuration of the BDR cluster.
This also simplifies configurations where the BDR cluster is spread over multiple data centers and only part of the database is replicated across data centers as each subgroup will automatically have new default replication set assigned to it.
- Multiple new monitoring views
Focused primarily on group level monitoring and in-progress monitoring on the apply side.
- Conflicts are now logged by default to `bdr.conflict_history`
Logging to a partitioned table with row level security to allow easier access to conflicts for application users.
- New conflict types `multiple_unique_conflicts` and `apply_error_ddl`
Allows continuing replication in more edge case situations
- Reduced lock levels for some DDL statements
Also, documented workarounds that help with reducing lock levels for multiple other DDL statements.
- Use best available index when applying update and delete
This can drastically improve performance for `REPLICA IDENTITY FULL` tables which don't have primary key.

Following are changes since 3.7.7.

25.5.3 Improvements

- Support Parallel Apply in EDB Advanced Server (EE)
- Increase progress reporting frequency when needed (BDR-436, BDR-522)
This helps speed up the performance of `VALIDATE CONSTRAINT` without DML locking.
- Change all BDR configuration options that are settable from SQL session to be settable by `bdr_superuser` rather than only Postgres superuser.
- Set `bdr.ddl_replication` to off in `bdr.run_on_all_nodes()` (BDR-445)
It's usually not desirable to replicate any DDL executed using the `bdr.run_on_all_nodes()` function as it already runs it on all nodes.
- Improve monitoring of transactions that are in progress on apply side (BDR-690, BDR-691)
Add query to `pg_stat_activity` when applying DDL and several additional fields to `bdr.subscription_summary` view which show LSN of latest received change, LSN of latest received commit, applied commit LSN, flushed LSN and applied timestamp.
This helps monitoring of replication progress, especially when it comes to large transactions.
- Add view `bdr.stat_activity`, similar to `pg_stat_activity` but shows BDR specific wait states.
- Allow batching inserts outside of the initial data sync
Improves performance of big data loads into existing BDR Group.
- Reduce the global lock level obtained by `DROP INDEX` from DML Global Lock to DDL Global Lock (BDR-652)

25.5.4 Resolved Issues

- Fix replication settings of several DDL commands
In general make sure that actual behavior and documented behavior for what's allowed, what's replicated and what locks are held during DDL replication match.
For example `TABLESPACE` related commands should not be replicated.
- Fix a race condition in concurrent join. (BDR-644, BDR-645)
Always create initially enabled subscription if the local node has already crossed the `PROMOTING` state.
- Set group leader for already held lock (BDR-418, BDR-291)
This solves "canceling statement due to global lock timeout" during some DDL operations when the writer already had open table before. This was especially problem when partitioning or parallel apply is involved.
- Progress WAL sender's slot based on WAL decoder input (BDR-567)
Without this, server could eventually stop working with single decoding worker.
- Switch to `TEMPORARY` replication slots in `bdr_init_physical` (BDR-191)
This ensures they are properly cleaned up after `bdr_init_physical` is done.
- Clean up `XID` progress records that are no longer required (BDR-436, BDR-532)
Reduces the size of the `xid` progress snapshot.
- Track `applied_timestamp` correctly in BDR Writer (BDR-609)
It was not updated in 3.7.7
- Fix creation of BDR Stream triggers on EPAS (BDR-581)
They used to be created as wrong trigger type.
- Improve error handling when options stored in LCR file and passed to `walsender` differ (BDR-551)

- Enable WAL decoder config only for top node group (BDR-566)
We only allow group configuration changes for top node group in general.
- Use “C” collation or “name” type for specific BDR catalog columns (BDR-561)
This solves potential index collation issues for BDR catalogs.
- Correct commit cancellation for CAMO
This fixes CAMO behavior when user cancels a query.
- Fix autopartition handling of tables with already existing partitions (BDR-668)
- Don’t cache relation with no remote id in BDRWrite (BDR-620)
Fixes replication breakage after some forms of TRUNCATE command.
- Craft upstream decoder slot name considering upstream dbname in wal decoder (BDR-460)
Fixes slot names used by wal decoder.
- Use correct BDR output options used by WAL decoder and WAL sender using LCR (BDR-714)
- Fix crash of monitor functions on a broken cluster. (BDR-580, BDR-696)
- Don’t show nonexistent slots for PARTED in bdr.node_slots view
- Drop Stream Trigger when dropping node (BDR-692)
This enables use of `bdr_init_physical` with Stream Triggers.
- Ensure we don’t segfault while handling a SIGUSR2 signal
Signals can come at any point in process lifetime so don’t make any assumptions about the current state.
- Handle concurrent drop of the table which can lead to missing autopartition rule
- Make sure we don’t crash when we get ERROR during handing of different ERROR
- Don’t send global xid to client if we are in background worker
There is nobody to send this.

25.5.5 Other Changes

- Allow session-level `bdr.xact_replication = off` when `bdr.permit_unsafe_commands` is on
Helps when using `pg_restore` to manually populate the database.
- Various larger documentaion improvements
- Throw nicer error when removing table from replication set if the table is not in the repset already (BDR-562)
- Allow `check_constraints` option again, but make sure it’s properly marked as deprecated (BDR-26)
Will be removed in BDR 4.0.
- Move the management of WAL senders when WAL decoder is enabled/disabled to manager process (BDR-612)
Managing them in consensus worker could negatively affect responsiveness of consensus subsystem.
- Check for interrupts in more places
Should reduce chance of runaway loops

25.6 BDR 3.7.7

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

25.6.1 Important Notes

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

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

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

25.6.2 Improvements

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

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

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

These are also added to the conflict history summary table.

Add local_time into bdr.conflict_history_summary local_time is the partition key of bdr.conflict_history, which we need to allow monitoring queries to execute efficiently.
- Add -node-group-name option to bdr_init_physical
Same as node_group_name in bdr.join_node_group - allows joining sub-group of a node.
- Store LCRs under directory named after WAL decoder slot (BDR-60)
Pglogical stores LCR in a directory named after the replication slot used to produce those.
- Various improvements in WAL decoder/sender coordination (BDR-232, BDR-335, BDR-342)

We now expose the information about WALDecoder waitlsn and let WALSender use that information to wait and signal the WALDecoder when the required WAL is available. This avoids the unnecessary polling and improves coordinator between the two.

- Single Decoder Worker GUC Option Changes. (BDR-222)
Changed `bdr.receive_logical_change_records` to `bdr.receive_lcr` and
`bdr.logical_change_records_cleanup_interval` to `bdr.lcr_cleanup_interval`
- Move most of the CAMO/Eager code into BDR (BDR-330)
Makes CAMO and Eager All Node less dependent on Postgres patches.
- Support the parallelization of initial sync.
When parallel apply is enabled, the initial sync during logical join will be parallellized as well.
- Deprecate `bdr.set_ddl_replication` and `bdr.set_ddl_locking`.

25.6.3 Resolved Issues

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

25.7 BDR 3.7.6

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

25.7.1 Important Notes

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

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

Upgrades are supported from BDR 3.6.25 in this release.

25.7.2 Improvements

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

25.7.3 Resolved Issues

- Make sure we have xid assigned before opening relation in writer
This should improve deadlock detection for parallel apply
- Check table oid in function drop_trigger (BDR-35)
Fixes crash when invalid oid was passed to the function.
- Fix application of older consensus snapshots (BDR-231)
We used to not handle missing group UUID correctly resulting in 3.7 node not being able to join 3.6 cluster.
- Readjust default truncate handling (BDR-25)
Don't take lock by default. While this can cause potential out of order truncation, it presents better backwards compatibility.
- Fix crash when OPTION clause is used in CREATE FOREIGN TABLE statement (EBC-37)
- Ensure that we don't send extra data while talking to node with old consensus protocol (BDR-135)
- Read kv_data part of consensus snapshot in mixed version group (BDR-130)
Both BDR 3.6. and 3.7 write this part of consensus snapshot but BDR 3.7 would only read it if the snapshot was also written by 3.7.
- Move bdr.constraint to EE script (EBC-36)
It's Enterprise Edition only feature so the catalog should only be installed with Enterprise Edition.
- Don't try to replicate GRANT/REVOKE commands on TABLESPACE and Large Objects
These objects are not replicated so trying to replicate GRANT and REVOKE would break replication.
- Make sure CAMO does not block replay progress (RT69493)
- Fix failed CAMO connection handling (RT69493, RM19924)
Correct the state machine to properly cleanup and recover from this failure and reset to the UNUSED & IDLE state.
- Don't accept Raft request from unknown nodes
Consensus leader should not accept raft request from nodes it does not know.
- Don't try to negotiate consensus protocol on unknown node progress (RT69779)
When node is forcefully dropped, we might still receive progress message from it. This has to gracefully ignore such message otherwise consensus could break in such situation.

25.7.4 Other Changes

- Remove code unsupported consensus protocols (BDR-86)

25.8 BDR 3.7.5

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

25.8.1 Important Notes

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

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

Upgrades are supported from BDR 3.6.22 in this release.

25.8.2 Improvements

- Reduce “now supports consensus protocols” log spam. (RT69557)
- Extend `bdr.drop_node` with a `node_state` check. (RM19280)
Adds a new argument ‘force’ to `bdr.drop_node`, defaulting to false, in which case the following additional check is performed: Via `bdr.run_on_all_nodes`, the current `node_state` of the node to be dropped is queried. If the node to be parted is not fully parted on all nodes, this now yields an error. The force argument allows to ignore this check. This feature also removes the “force” behavior that cascade had before, now we have two distinct options, one to skip sanity checks (force) and one to cascade to dependent objects (cascade).
- Deprecate `pg2q.enable_camo` (RM19942, RT69521)
The parameter has been changed in 3.7 to the new `bdr.enable_camo`.
- Add new parameter `detector_args` to `bdr.alter_table_conflict_detection` (RT69677)
Allow additional parameters for individual detectors. Currently just adds `atttype` for `row_version` which allows using `smallint` and `bigint`, not just the default integer for the column type.
- Add `bdr.raft_leadership_transfer` (RM20159)
Promote a specific node as the Raft leader. Per Raft paper, transferring leadership to a specific node can be done by the following steps:
 - the current leader stops accepting new requests
 - the current leader sends all pending append entries to the designated leader
 - the current leader then forces an election timeout on the designated leader, giving it a better chance to become the next leaderThe feature pretty much follows that outline. Instead of sending append entries just to the designated leader, we send it to all nodes as that also acts as a heartbeat. That should ensure that no other node times out while the current leader delegating power to the designated node. We also check status of the designated node and don’t accept the request if the node is not an active node or if it doesn’t have voting rights.
- Implement the concept of `subscriber-only` nodes
These are wholly joined nodes, but they don’t ever send replication changes to other BDR nodes in the cluster. But they do receive changes from all nodes in the cluster (except, of course the other subscriber-only nodes). They do not participate in the RAFT voting protocol, and hence their presence (or absence) does not determine RAFT leader election. We don’t need to create any replication slots on these nodes since they don’t send replication changes. Similarly, we don’t need to create any subscriptions for these nodes on other BDR nodes. We implement this by defining a new type of BDR node group, called “subscriber-only” group. Any node supposed to be a subscriber-only node should join this node group instead of the top level BDR group. Of course, someone needs to create the subscriber-only BDR nodegroup first. The feature does not attempt to create it automatically.
- Improve DDL replication support for PostgreSQL 13
The `ALTER STATISTICS` and `ALTER TYPE ... SET` commands are now supported.

25.8.3 Resolved Issues

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

25.9 BDR 3.7.4

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

25.9.1 Important Notes

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

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

Upgrades are supported from BDR 3.6.22 in this release.

25.9.2 Improvements

- Add support for PostgreSQL 13
- Extend `bdr.get_node_sub_receive_lsn` with an optional `committed` argument
The default behaviour has been corrected to return only the last received LSN for a committed transaction to apply (filtered), which is the original intent and use of the function (e.g. by HARP). Passing a `false` lets this function return the unfiltered most recent LSN received, matching the previous version's behavior. This change is related to the hang in `bdr.wait_for_apply_queue` mentioned below.
- Error out if INCREMENT BY is more than gallo chunk range (RM18519) The smallint, int and bigint gallo sequences get 1000, 1000000, 1000000000 values allocated in each chunk respectively. We error out if the INCREMENT value is more than these ranges.
- Add support for validating constraints without a global DML lock (RM12646)
The DDL operation `ALTER TABLE ... ADD CONSTRAINT` can take quite some time due to the validation to be performed. BDR now allows deferring the validation and running the `ALTER TABLE ... VALIDATE CONSTRAINT` part without holding the DML lock during the lengthy validation period.

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

25.9.3 Resolved Issues

- Correct a hang in `bdr.wait_for_apply_queue` (RM11416, also affects CAMO)
Keepalive messages possibly move the LSN forward. In an otherwise quiescent system (without any transactions processed), this may have led to a hang in `bdr.wait_for_apply_queue`, because there may not be anything to apply for the corresponding PGL writer, so the `apply_lsn` doesn't ever reach the `receive_lsn`. A proper CAMO client implementation uses `bdr.logical_transaction_status`, which in turn uses the affected function internally. Thus a CAMO switch- or fail-over could also have led to a hang. This release prevents the hang by discarding LSN increments for which there is nothing to apply on the subscriber.
- Allow consensus protocol version upgrades despite parted nodes (RM19041)
Exclude already parted nodes from the consensus protocol version negotiation, as such nodes do not participate in the consensus protocol any more. Ensures the newest protocol version among the set of active nodes is used.
- Numerous fixes for gallo sequences (RM18519, RM18512) The “nextval” code for gallo sequences had numerous issues:
 - Large INCREMENT BY values (+ve or -ve) were not working correctly

- Large CACHE values were not handled properly
- MINVAL/MAXVAL not honored in some cases The crux of the issue was that large increments or cache calls would need to make multiple RAFT fetch calls. This caused the loop retry code to be invoked multiple times. The various variables to track the loops needed adjustment.
- Fix tracking of the last committed LSN for CAMO and Eager transactions (RM13509)
The GUC `bdr.last_committed_lsn` was only updated for standard asynchronous BDR transactions, not for CAMO or Eager ones.
- Fix a problem with NULL values in `bdr.ddl_epoch` catalog (RM19046, RM19072)
Release 3.7 added a new `epoch_consumed_lsn` column to `bdr.ddl_epoch` catalog. Adding a new column would set the column value to NULL in all existing rows in the table. But the code failed to handle the NULL values properly. This could lead to reading garbage values or even memory access errors. The garbage values can potentially lead to global lock timeouts as a backend may wait on a LSN which is far into the future.
We fix this by updating all NULL values to '0/0' LSN, which is an invalid value representation for LSN. The column is marked NOT NULL explicitly and the code is fixed to never generate new NULL values for the column.
- Corrections for upgrading from BDR 3.6.22
Properly migrate subscription writer and conflict handlers from PGLogical, where this information used to be with BDR 3.6. Ensure `bdr.conflict_history` is handled properly after an upgrade.
- Fix JOINING state handling on consensus request timeout (RT69076)
The timeout during JOINING state handling could result in node unable to join the BDR group. The retry logic now handles this state correctly.
- Validate inputs to `replication_set_remove_table` (RT69248, RM19620)
- Handle missing column gracefully for `ALTER COLUMN TYPE` (RM19389, RT69114)
Throw the standard ERROR rather than crashing when this happens.
- Fix memory handling of a tuple slot during conflict lookup (RM18543)
No longer crashes when the found tuple is logged into conflict log table.
- Fix local node cache invalidation handling (RM13821)
Previously BDR might not notice node creation or node drop due to race conditions, and would chose wrong behavior inside user backend.

25.10 BDR 3.7.3

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

25.10.1 Important Notes

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

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

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

25.10.2 Improvements

- **Parallel Apply (RM6503)**
Using the new infrastructure in pglogical 3.7.3, add support for parallel writers.
The defaults are controlled by same pglogical configuration options (and hence this feature is currently off by default)
The number of parallel writers can be changed per group using the `num_writers` parameter of the `bdr.alter_node_group_config()` administration interface.
- **resynchronize_table_from_node()** works with the generated columns (RM14876)
It copies all the columns except the generated columns from remote node and computes the generated column values locally.
- **resynchronize_table_from_node()** freezes the table on target node (RM15987) When we use this function the target table is truncated first and then copied into on the destination node. This activity additionally FREEZES the tuples when the resync happens. This avoids a ton of WAL activity which could potentially happen when hint bit related I/O+WAL would come into the picture in the future on this destination node.
- Allow use of CRDTs on databases with BDR extension installed but without any node (RM17470). Earlier restoring CRDT values on a node with BDR extension, but without any node, would have failed with an ERROR as the CRDT data type queries for the node identifier. It is now fixed by storing an `InvalidOid` value when the node identifier is not available. If the node is subsequently added to a BDR cluster and when the CRDT value is updated, `InvalidOid` will be replaced by a proper node identifier as part of the UPDATE operation.
- Add consistent KV Store implementation for the use by the HARP project (RM17825)
This is not meant for direct user consumption, but enables the HARP to work with BDR without additional consensus setup.

25.10.3 Resolved Issues

- Re-add the “local_only” replication origin (RT68021)
Using `bdr_init_physical` may have inadvertently removed it due to a bug that existing up until release 3.6.19. This release ensures to recreate it, if it’s missing.
- Handle NULL arguments to `bdr.alter_node_set_log_config()` gracefully (RT68375, RM17994)
The function caused segmentation fault when the first argument to this function is NULL. It is now fixed to provide an appropriate error message instead.
- Fix MAXVALUE and MINVALUE with gallo sequences (RM14596)
While fetching values in advance, we could have reached the limit. Now we use only the values that we fetched before reaching the limit.
- Optionally wait for replication changes triggered by prior epoch (RM17594, RM17802) This improves handling of multiple concurrent DDL operations across the BDR Group which would previously result in global lock timeout, but now are allowed to pass as long as the replication lag between nodes is not too large.
- **resynchronize_table_from_node()** now correctly checks membership of the resynchronized table in replication sets subscribed by the target node (RM17621) This is important in order to not allow unprivileged users to copy tables that they don’t have otherwise ability to access.
- Allow new group creation request to work after previous attempt has failed (RM17482)
Previously, the new requests would always fail in some setups until BDR was completely removed from the node and reinstalled if the initial group creation has failed.
- Lower the CPU consumption of consensus worker when Autopartition feature is used (RM18002)
- Fix memory leak during initial data synchronization (RM17668)

- Fix `update_recently_deleted` conflict detection (RM16471)
This conflict was not detected correctly in 3.7.2.
- Check the options when altering a gallo sequence (RM18301, RT68470) Gallo sequences do not accept some modifications, warn the user in case not allowed options are used.
- Make sure `bdr_wait_slot_confirm_lsn` is waiting for all slots (RM17478)
This function used to skip some of the slots when checking if downstream has replicated everything.
- Improve `PART_CATCHUP` node state handling (RM17418)
Resolves cases where node state would stay `PART_CATCHUP` forever due to race condition between nodes.
- Make the consensus process more resilient when there are missing parted nodes
Don't fail when trying to update a node's state to `PARTED` and the node no longer exists.
- Remove `--recovery-conf` argument from `bdr_init_physical` (RM17196)
It didn't work previously anyway and PostgreSQL12 does not longer have `recovery.conf`.

25.10.4 Other Improvements

- Enable `bdr.truncate_locking` by default
This is needed for `TRUNCATE` operations to always produce consistent results when there is concurrent DML happening in the BDR Group. This was missed by previous beta.
- Create a virtual sequence record on other nodes RM16008 If we create a gallo sequence and try to use its value in the same transaction block, then because it does not exist yet on other nodes, it used to error out with “could not fetch next sequence chunk” on the other nodes. We solve this by creating a virtual record on the other nodes.
- Significant improvements to the language in documentation.

25.11 BDR 3.7.2

This is a beta release of the BDR 3.7.

25.11.1 Important Notes

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

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

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

25.11.2 The Highlights of BDR 3.7

- Parallel Apply
Allows configuring number of parallel writers that apply the replication stream.
- AutoPartition
See [AutoPartition](#) for details.
- Support `CREATE TABLE ... AS` statement (RM9696)
This feature is now supported in Enterprise Edition only.

- New ability to define BDR sub-groups in order to better represent physical configuration of the BDR cluster. This also simplifies configurations where the BDR cluster is spread over multiple datacenters and only part of the database is replicated across datacenters as each subgroup will automatically have new default replication set assigned to it.
- Conflicts are now logged by default to `bdr.conflict_history`
Logging to a partitioned table with row level security to allow easier access to conflicts for application users.
- New conflict type `multiple_unique_conflicts`
Allows resolution of complex conflicts involving multiple UNIQUE constraints for both INSERT and UPDATE.
- Merge views `bdr.node_replication_rates` and `bdr.node_estimate` into `bdr.node_replication_rates`. `bdr.node_estimate` has been removed (RM13523)
- Don't replicate REINDEX command, now treated as a maintenance command
- Various other changes to default settings

25.11.3 Other Improvements

- Optional monitoring tables for describing node connections and geographical distribution
- Add `bdr.resynchronize_table_from_node` function (RM13565, RM14875)
This function resynchronizes the relation from a remote node. This acquires a global DML lock on the relation, truncates the relation locally, and copies data into it from the remote node. The relation must exist on both nodes with the same name and definition.
- Add a function `bdr.trigger_get_origin_node_id` to be used in conflict triggers (RM15105, RT67601)
This will enable users to define their conflict triggers such that a trusted node will always win in case of DML conflicts.
- Extend `bdr.wait_for_apply_queue` to wait for a specific LSN (RM11059, RT65827)
- Add committed LSN reporting via `bdr.last_committed_lsn` (RM11059, RT65827)
- BDR now accepts also URI in connection strings (RM14588)
We can now specify also the format URI “postgresql://...” for the connection string.

25.11.4 Resolved Issues

- Resilience against `idle_in_transaction_session_timeout` (RM13649, RT67029, RT67688)
Set `idle_in_transaction_session_timeout` to 0 so we avoid any user setting that could close the connection and invalidate the snapshot.
- Correct parsing of BDR WAL messages (RT67662)
In rare cases a DDL which is replicated across a BDR cluster and requires a global lock may cause errors such as “invalid memory alloc request size” or “insufficient data left in message” due to incorrect parsing of direct WAL messages. The code has been fixed to parse and handle such WAL messages correctly.
- Fix locking in ALTER TABLE with multiple sub commands (RM14771)
Multiple ALTER TABLE sub-commands should honor the locking requirements of the overall set. If one sub-command needs the locks, then the entire ALTER TABLE command needs it as well.

25.12 BDR 3.6.19

This is a security and maintenance release for BDR 3.6 which includes also includes various minor features.

25.12.1 Resolved Issues

- **SECURITY:** Set `search_path` to empty for internal BDR SQL statements (RM15373)
Also, fully qualify all operators used internally. BDR is now protected from attack risks identified in CVE-2018-1058, when the user application avoids the insecure coding practices identified there. See BDR Security chapter for further explanation.
 - **SECURITY:** Raise required privileges for BDR admin functions (RM15542)
When executed by roles other than superuser or `bdr_superuser`:
 - `bdr.alter_table_conflict_detection` needs table owner
 - `bdr.column_timestamps_enable` needs table owner
 - `bdr.column_timestamps_disable` needs table owner
 - `bdr.drop_trigger` needs table owner
 - `bdr.alter_sequence_set_kind` needs sequence owner
 - `bdr.global_lock_table` needs UPDATE, DELETE or TRUNCATE (like LOCK TABLE)
 - `bdr.create_conflict_trigger` needs TRIGGER permission on the table and EXECUTE permission on the function
 - `bdr.create_transform_trigger` needs TRIGGER permission on the table and EXECUTE permission on the function
- A new GUC `bdr.backwards_compatibility` allows to skip this newly introduced check for existing clients requiring the former behavior.
- Resolve a hang possible after multiple global lock releases (RT67570, RM14994)
A bug in the code path for releasing a global lock after a timeout led to overriding the backend's PID with a value of -1, also showing up in the `waiters` list of `bdr.global_locks`. This in turn crippled the `waiters` list and ultimately led to an infinite loop. This release fixes the override, which is the original cause of this hang and correctly removes entries from the lock wait list.
 - Correct parsing of BDR WAL messages (RT67662)
In rare cases a DDL which is replicated across a BDR cluster and requires a global lock may cause errors such as “invalid memory alloc request size” or “insufficient data left in message” due to incorrect parsing of direct WAL messages. The code has been fixed to parse and handle such WAL messages correctly.
 - Fix locking in ALTER TABLE with multiple sub commands (RM14771) Multiple ALTER TABLE sub-commands should honor the locking requirements of the overall set. If one sub-command needs the locks, then the entire ALTER TABLE command needs it as well.
 - Fix bug in example of ALTER TABLE ... ADD COLUMN workaround (RT67668)
Explain why `bdr.global_lock_table()` is needed to avoid concurrent changes that cause problems, in that case.
 - Fix a hang after promotion of a physical standby (RM15728)
A physical standby promoted to a BDR node may have failed to start replicating due to the use of stale data from an internal catalog cache.
 - Fix crash when `bdr.trigger_get_type()` is called by itself (RM15592) Calling `bdr.trigger_get_type()` outside a streaming trigger function would cause a crash. Fixed the function to return NULL when called outside a streaming trigger function.

25.12.2 Improvements

- `bdr.trigger_get_origin_node_id()` allows preferred-node resolution (RM15105, RT67601)
Some customers have a requirement to resolve conflicts based upon the node that is the source of the change. This is also known as trusted source, trusted site or AlwaysWins resolution. Previous versions of BDR allowed these mechanisms with 2 nodes; this new function allows this option with any number of nodes. Examples are documented.
- BDR now accepts URIs in connection strings (RM14588)
All connection strings can now use the format URI “`postgresql://...`”
- New function `bdr.resynchronize_table_from_node()` (RM13565, RT67666, RT66968) allows a single table to be truncated and then resynced from a chosen node, while holding a global dml lock. This allows a table to be resynchronized following a data divergence or data corruption without needing to regenerate the whole node. Foreign Keys are removed and re-enabled afterwards.
- Improve filtering of changes made by explicitly unreplicated transactions (RM15557)
Previously changes made by transactions using `bdr.xact_replication = off` or by `bdr.difference_fix` transactions would be sent to the remote node, generating spurious conflicts and wasting effort. Changes are now filtered on the source node instead, improving performance.
- Initial and periodic transaction status checks use async libpq (RM13504) (EE)
With CAMO enabled, the status of in-flight transactions is checked against a partner node. This uses an standard Postgres connection via libpq, which used to block the PGL manager process. This release changes the logic to use asynchronous libpq to allow the PGL manager to perform other tasks (e.g. process Raft messages) while that status check is performed. This reduces chances of timeouts or deadlocks due to a more responsive PGL manager process.
- Additional message fields assist diagnosis of DDL replication issues (RM15292)
- Clarify documentation regarding privileges required for BDR users (RT67259, RM15533)

25.13 BDR 3.6.18

This is a maintenance release for BDR 3.6 which includes minor features as well as fixes for issues identified previously.

25.13.1 Improvements

- Add `synchronize_structure` option to `join_node_group` (RM14200, RT67243)
New `synchronize_structure` option can be set to either ‘all’ or ‘none’, which either synchronizes the whole schema or copies no DDL. This allows for rolling application schema upgrades to be performed with a user-managed schema (DDL) change step.
- Make `bdr*difference_fix**` functions use pre-created local origin (RM14189)
The `bdr*difference_fix**` family of functions used to create a local origin to carry out conflict fixes. We now pre-create “`bdr_local_only_origin`” local origin at extension creation time. This same local origin is used by the above functions now.
- Adjust monitored values in `bdr.monitor_group_versions()` (RM14494)
We no longer report CRITICAL when pglogical version different to bdr version, which is actually not important. We now report WARNING if BDR editions differ between nodes.
- Substantial formatting corrections and spelling check of documentation

25.13.2 Resolved Issues

- Fix node join so it uses only `bdr_superuser` permissions (RM14121, RT67259)
This affects the `join_target_dsn` connection of the `join_node_group` function, which has been fixed to work with only `bdr_superuser` right for the role used to connect.
- GRANT EXECUTE on `bdr.show_subscription_status` TO `bdr_real_all_stats` (RT67360, RM14624)
This allows both `bdr_read_all_stats` and `bdr_monitor` roles to access the `bdr.subscription_summary` view
- Fix failure of `bdr_init_physical` to copy data columns using BDR types (RM14522)
`bdr_init_physical` now uses `bdr.drop_node()` rather than `DROP EXTENSION`, which caused all columns using BDR datatypes such as CRDTs to be silently dropped from tables.
- Fix failure in 3.6.17 upgrade script caused by views referencing CRDTs (RT67505)
Upgrade script now executed only on tables and mat views. Upgrade failure may give a spurious error such as “ERROR: BDR global lock manager not initialized yet”
- Set non-join subscriptions to CATCHUP state rather than INIT state at startup
Avoids a rare but possible case of copying metadata twice during node join.
- Fix lookup for a gallo sequence when BDR catalogs are absent. (RT67455, RM14564)
This might cause a query on a sequence to throw an error like “cache lookup failed for relation ...” when `bdr` library is added to `shared_preload_libraries` but BDR extension is not created.
- Allow ALTER TABLE ALTER COLUMN with BDR loaded but not initialized (RM14435)
With the BDR extension loaded, but no local BDR node created, the DDL replication logic now still allows execution of an ALTER TABLE ALTER COLUMN operation.
- LOCK TABLE warning not shown when BDR node is not created (RM14613)
Assess LOCK TABLE statement does not show when `bdr.assess_lock_statement` is set to a value other than ‘ignore’ until BDR node is created.
- Prevent a NULL dereference in `consensus_disable` (RM14618)
`bdr.consensus_disable` expected the consensus process to be running. Fix it to prevent a segfault if that’s not the case when the function is called.

25.14 BDR 3.6.17

This is a maintenance release for BDR 3.6 which includes minor features as well as fixes for issues identified previously.

25.14.1 Improvements

- Allow ALTER TABLE ALTER COLUMN TYPE with rewrite when not replicating DDL (EE) (RM13244)
In some cases, in controlled DBA environments, it is possible to change the type of a column to an implicitly 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. We allow concurrent activity on this table on other nodes during the rewrite. Also note that such ALTER commands cannot be run within transaction blocks.
- Add conflict logging configuration view (RM13691, RT66898)
Add `bdr.node_log_config` view that shows information on the conflict logging configuration.
- Add new group monitoring views and functions (RM14014)
These views and functions report the state of the BDR installation, replication slots and consensus across all nodes in the BDR group.

- Add current state of DDL replication related configuration parameters to log context (RM13637)
Improves troubleshooting.

25.14.2 Resolved Issues

- Don't drop existing slot for a joining node (RM13310, RT67289, RT66797)
This could have caused inconsistencies when node was joined using `bdr_init_physical` because it precreated the slot for new node which was supposed to be reused during join, instead it was dropped and recreated. We now keep the slot correctly which ensures there are no inconsistencies.
- Fix restart of CAMO node despite missing partner node (EE) (RM13899, RT67161)
Prevent an error looking up the BDR node configured as a CAMO origin. In case the node got dropped, it does not exist, but might still be configured for CAMO.
- Fix locking in `bdr.column_timestamps_enable()` (EE) (RT67150)
Don't hold same transaction and session level locks otherwise PREPARE, CAMO and Eager replication can't work for transactions where this is used.
- Don't try to apply BDR conflict resolution to PGL-only subscriptions (RT67178)
BDR should only be active on BDR subscriptions, not pglogical ones.
- Let the CAMO partner return the final decision, once learned (RM13520)
If an origin node switches to Local mode, temporarily dropping CAMO protections, it's possible for the CAMO partner to provisionally abort the transaction internally, but actually commit it eventually (to be in sync with the origin node). In earlier releases, this was not recorded leading to the status query function to continue to return an "aborted" result for the transaction. This release allows the final commit decision to override the provisional abort internally (catalog table `bdr.node_pre_commit`).
- Make CLCD/CRDT data types properly TOAST-able (EE) (RM13689)
CLCD/CRDT data types were defined as using PLAIN storage. This can become an issue with a table with too many columns or if a large number of nodes are involved. This is now solved by converting these data types to use EXTENDED storage thus allowing for large sized values.
- Ensure duplicate messages are not received during node promotion (RM13972) Send a watermark from join source to the joining node during catchup phase of join to ensure it learns about current replication positions of all other nodes even if there are no data to forward from them during the catchup. Otherwise we might ask for older lsns during the promotion and receive duplicate messages and fail the join.
- Automatically disable CAMO for non-transactional DDL operations (EE)
Several DDL operations are not allowed within a transaction block and as such cannot reasonably benefit from the protection that CAMO offers. Automatically disable CAMO for these, so as to avoid "cannot PREPARE" errors at COMMIT time.
- Fix errors when `bdr.move_group_slot_all_nodes` is called with no BDR node present in the database (RT67245)
Allows setting up physical standbys of future BDR master before creating the BDR node.
- Make sure table has a PRIMARY KEY when CLCD is turned on (EE)
This is a sanity check that prevents user from enabling CLCD on tables without a PRIMARY KEY as that would break the conflict detection for such tables.

25.15 BDR 3.6.16

BDR 3.6.16 is the sixteenth minor release of the BDR 3.6 series. This release includes minor new features as well as fixes for issues identified previously.

25.15.1 Improvements

- Add `bdr.alter_table_conflict_detection()` (RM13631)
This function unifies the UI for changing conflict detection method for individual tables. Allows choice between origin based, row_version based and column level based (EE-only) conflict detection using same interface. The old functions are still supported, although they should be considered deprecated and will be removed in BDR 3.7.
- Add `bdr.default_conflict_detection` configuration option (RM13631)
Related to the above `bdr.alter_table_conflict_detection()` function, the new configuration option allows setting the default conflict detection method for newly created tables.
- Change how forced part node works (RM13447)
Forced node part will now first try to get consensus for parting and only do the local change if the consensus fails or if it's called for node which already started consensus based part process but the process has stuck on one of the steps.
- Automatically drop `bdr-enterprise` extension when dropping the `bdr` extension (RM13703)
This improves usability when trying to drop the `bdr` extension without cascade, which is useful for example when user wants to keep the pglogical node associated with BDR.
- Improve error reporting when joining node with same name as existing active node (RM13447, RT66940)
The previous error message was confusing as it made it seem like BDR does not allow node name reuse at all (which it does).
- Set `application_name` in `bdr_init_physical`
Helps when diagnosing issues with this tool.
- Improve documentation of `ALTER TABLE` limitations (RM13512, RM13244, RT66940)
Including new workaround for changing length of varchar columns.

25.15.2 Resolved Issues

- Fix `pg_dump` for BDR galloc sequences (RM13462, RT67051)
Galloc sequences internally store extra data in sequence heap; BDR now hides the extra data from `SELECT`s so that queries on the sequence (which can be normal user query or a query from `pg_dump` for example) only show the usual sequence information.
- Fix enforcement of `REPLICA IDENTITY FULL` for CLCD
Advanced conflict-handling approaches (CLCD, CRDT) require the table to have `REPLICA IDENTITY FULL`. However due to how the features initially evolved independently, this was not enforced (and documented) properly and consistently. We now correctly enforce the `REPLICA IDENTITY FULL` for CLCD for every table.
- Fix node name reuse of nodes which were used as join sources for other existing nodes in a BDR group (RM12178, RM13447)
The source nodes have special handling so we need to make sure that newly joining node is not confused with node of same name that has been parted.
- Apply local states for existing nodes on newly joining node (RT66940)

Otherwise decision making in during the join process might use wrong state information and miss some tasks like slot creation or subscription creation.

- Correctly clean node-level log filters and conflict resolver configuration (RM13704)
This solves issues when trying to drop BDR node without dropping associated pglogical node and later recreating the BDR node again.
- Prevent a segfault in Raft on the parted BDR node (RM13705)

25.16 BDR 3.6.15

BDR 3.6.15 is the fifteenth minor release of the BDR 3.6 series. This release includes minor new features as well as fixes for issues identified previously.

25.16.1 Improvements

- Keep a permanent log of all resolved CAMO decisions (RM12712)
Record every decision taken by the CAMO partner when queried by `bdr.logical_transaction_status`, i.e. in the failover case.
- Add functions for enabling/disabling row version tracking (RM12930)
Easier to use and less error prone interface than manually adding column and trigger.
- Add `currval()` and `lastval()` support for `timeshard` and `galloc` sequences (RM12059)
- Add `pglogical.min_worker_backoff_delay` setting to rate limit background worker re-launches, and `pglogical.worker_tasks` diagnostic view for background worker activity. See pglogical 3.6.15 release notes and documentation for details.

25.16.2 Resolved Issues

- Prevent buffer overrun when copying a TOAST column value inside the walsender output plugin (RT66839)
This fixes issue that resulted in walsender crashes with certain types of workloads which touch TOASTed columns.
- Fix “type bdr.column_timestamps not found” error when bdr-enterprise extension is not installed when bdr enterprise library is in `shared_preload_libraries` (RT66758, RM13110)

25.17 BDR 3.6.14

BDR 3.6.14 is a critical maintenance release of the BDR 3.6 series. This release includes major fixes for CAMO and other features as well as minor new features.

25.17.1 Improvements

- Add `bdr.camo_local_mode_delay` to allow throttling in CAMO Local mode (RM12402)
Provides a simple throttle on transactional throughput in CAMO Local mode, so as to prevent the origin node from processing more transactions than the pair would be able to handle with CAMO enabled.
- Add `bdr.camo_enable_client_warnings` to control warnings in CAMO mode (RM12558)
Warnings are emitted if an activity is carried out in the database for which CAMO properties cannot be guaranteed. Well-informed users can choose to disable this if they want to avoid such warnings filling up their logs.
- Warn on unrecognized configuration settings
- Move ‘loading BDR’ message earlier in startup messages
- Significantly enhance docs for Row Version Conflict Detection (RT66493)
- Clarify docs that NOTIFY is not possible with CAMO/Eager
- Add `global_lock_request_time`, `local_lock_request_time` and `last_state_change_time` columns to `bdr.global_locks` view for lock monitoring and diagnostic use.
- Add SQL functions for export/import of consensus snapshot (RM11433)
These functions allow for manual synchronization of BDR system catalogs in case of corruption or user mistake.

25.17.2 Resolved Issues

- UPDATES skipped on the partner node because `remote_commit_ts` set incorrectly (RM12476)
Commit timestamps were unset in some CAMO messages, leading to losing last-update-wins comparisons that they should have won, which meant some UPDATES were skipped when an UPDATE happened concurrently from another master. This doesn’t occur normally in an AlwaysOn cluster, though could occur if writes happen via a passive master node.
- Only resolve those prepared transactions for which controlling backend is gone (RM12388)
This fixes a race condition between the pglogical manager process and the user backend running a CAMO transaction. A premature attempt by the manager process to resolve a prepared transaction could lead to the transaction getting marked as aborted on the partner node, whereas the origin ends up committing the transaction. This results in data divergence. This fix ensures that the manager process only attempts to resolve prepared transactions for which the controlling user backend has either exited or is no longer actively managing the CAMO transaction. The revised code also avoids taking ProcArrayLock, reducing contention and thus improving performance and throughput.
- Prevent premature cleanup of commit decisions on a CAMO partner. (RM12540)
Ensure to keep commit or abort decisions on CAMO or Eager All Node transactions in `bdr.node_pre_commit` for longer than 15 minutes if there is at least one node that has not learned the decision and may still query it. This eliminates a potential for inconsistency between the CAMO origin and partner nodes.
- Resolve deadlocked CAMO or Eager transactions (RM12903, RM12910)
Add a `lock_timeout` as well as an abort feedback to the origin node to resolve distributed deadlocking due to conflicting primary key updates. This also prevents frequent restarts and retries of the PGL writer process for Eager All Node and sync CAMO transactions.
- Fix potential divergence by concurrent updates on toasted data from multiple nodes (RM11058)
This can occur when an UPDATE changes one or more toasted columns, while a concurrent, but later UPDATE commits on a different node. This occurs because PostgreSQL does not WAL log TOAST data if it wasn’t changed by an UPDATE command. As a result the logically decoded rows have these columns marked as

unchanged TOAST and don't contain the actual value. Fix is handled automatically on BDR-EE, but on BDR-SE additional triggers need to be created on tables that publish updates and that have toastable data (this is also done automatically). The additional check has a small but measurable performance overhead. Logged data will increase in affected cases only. We recommend tuning `toast_tuple_target` to optimize storage. Tables with `REPLICA IDENTITY FULL` are not affected by this issue or fix.

- Properly close connections after querying camo partner to avoid leak. (RM12572)
- Correct `bdr.wait_for_apply_queue` to respect the given LSN (RM12552)
In former releases, the `target_lsn` argument was overridden and the function acted the same as if no `target_lsn` had been given.
- Ignore progress messages from unknown nodes (RT66461)
Avoids problems during node parting.
- Make `bdr.xact_replication` work with `ALTER TABLE` and parallel query (RM12489)

25.18 BDR 3.6.12

BDR 3.6.12 is the twelfth minor release of the BDR 3.6 series. This release includes minor new features as well as fixes for issues identified previously.

25.18.1 Improvements

- Apply `check_full_row` on `DELETE` operations (RT66493)
This allows detection of `delete_recently_updated` conflict even if the `DELETE` operation happened later in wall-clock time on tables with full row checking enabled.
- Improve Global DML lock tracing
Add more information to the Global DML Lock trace to help debugging global locking issues more effectively.
- Validate replication sets at join time. (RM12020, RT66310)
Raise an `ERROR` from `bdr.join_node_group()` if the joining node was configured to subscribe to non-default replication sets by using `bdr.alter_node_replication_sets()` before join but some of the subscribed-to replication sets are missing.

On prior releases the joining node might fail later in the join process and have to be force-parted. Or it might appear to succeed but join with empty tables.

25.18.2 Resolved Issues

- Fix crash in `bdr.run_on_all_nodes` (RM12114, RT66515)
Due to incorrect initialization the `bdr.run_on_all_nodes` could have previously crashed with segmentation fault in presence of `PARTED` nodes.
- Don't broadcast the epoch consumed WAL messages (RM12042)
Broadcasting the message to all nodes could result in some nodes moving the Global DDL Lock Epoch forward in situations where it wasn't safe to do so yet, resulting in lowered protection against concurrent DML statements when running a statement that requires a Global DML Lock.
- Fix global locking on installations with multiple BDR nodes on single PostgreSQL instance
The global locking could get spurious timeouts because the lock messages contained wrong node id if there were more than one BDR node on a single PostgreSQL instance.

- Fix typos in some example SQL in docs

25.19 BDR 3.6.11

BDR 3.6.11 is the eleventh minor release of the BDR 3.6 series. This release includes minor new features as well as fixes for issues identified previously.

25.19.1 Improvements

- Support APIs for PostgreSQL 11.6
- Allow the use of “-”(hyphen) character in the node name (RM11567, RT65945)
If a pglogical3 node would have been created with a hyphen in the node name BDR couldn't create the node on that database.
- Don't generate `update_origin_change` conflict if we know the updating node has seen the latest local change (RM11556, RT66145)
Reduces conflict resolution overhead and logging of `update_origin_change` when the conflict can be shown to be false-positive. This does not completely remove false-positives from `update_origin_change` but reduces their occurrence in presence of UPDATES on older rows. This reduces conflict log spam when origin changes for older rows. Also, conflict triggers will be called significantly fewer times.
- Extend `bdr.wait_for_apply_queue` to wait for a specific LSN (RM11059, RT65827)
- Add new parameter `bdr.last_committed_lsn` (RM11059, RT65827)
Value will be reported back to client after each COMMIT, allowing applications to perform causal reads across multiple nodes.
- Add status query functions for apply and received LSN (RM11059, RM11664)
New functions `bdr.get_node_sub_receive_lsn` and `bdr.get_node_sub_apply_lsn` simplify fetching the internal information required for HAproxy health check scripts.
- Add `sum()` and `avg()` aggregates for CRDT types (RM11592, RT66168)
- Speed up initial synchronization of replication slots on physical standby (RM6747)
- Add `bdr.pg_xact_origin` function to request origin for an xid (RM11971)
- Add `bdr.truncate_locking` configuration option which sets the TRUNCATE command's locking behavior (RT66326)
This configuration option determines whether (when `true`) TRUNCATE obeys the `bdr.ddl_locking` setting which is the new, safe behavior or if (when `false`, the default) never does any locking, which is the old, potentially unsafe behavior.
- Allow conflict triggers to see commit timestamp of `update_recently_deleted` target rows (RM11808, RT66182)

25.19.2 Resolved Issues

- Add hash/equality opclass for the `column_timestamps` data type (RT66207)
REPLICA IDENTITY FULL requires comparison of all columns of a tuple, hence `column_timestamps` data type must support equality comparisons.
- Correct conflict docs for BDR-EE (RT66239, RM9670)
Changes made in BDR3.5 were not correctly reflected in conflict docs
- Don't check protocol version for galloc sequences during initial sync (RM11576, RT65660)
If galloc sequences already exist, `bdr_init_physical` doesn't need to recheck protocol versions.
- Fix galloc sequence chunk tracking corruption on lagging nodes (RM11933, RT66294)
In presence of node with lagging consensus the chunk tracking table would diverge on different nodes which then resulted in wrong chunks being assigned on consensus leader change. As a result node might start generating already used sequence numbers. This fix ensures that the table never diverges.
- Fix galloc sequence local chunk information corruption (RM11932, RT66294)
Make sure we correctly error out when in all cases request of new chunk has failed, otherwise we might assign bogus chunks to the sequence locally which would result in potentially duplicate sequence numbers generated on different nodes.
- Fix a case where the consensus worker event loop could stall in the message broker when trying to reconnect to an unreachable or unresponsive peer node by being more defensive about socket readability/writability checks during the libpq async connection establishment phase. (RM11914)

This issue is most likely to arise when a peer node's underlying host fails hard and ceases replying to all TCP requests, or where the peer's network blackholes traffic to the peer instead of reporting a timely ICMP Destination Unreachable message.

Effect of the issue on affected nodes would result in operations which require consensus to either stall or not work at all - those include: DDL lock acquisition, Eager transaction commit, calling `bdr.get_consensus_status()` function, galloc sequence chunk allocation, leader election and BDR group slot advancing. This could have been visible to users as spurious lock timeout errors or increased lag for the BDR group slot.
- Fix a race condition with global locks and DML (RM12042)
Prevent mismatching ordering of lock operations against DML with three or more concurrently writing nodes. This allows to properly protect a TRUNCATE against concurrent DML from multiple writer nodes.
- Repeat reporting of `local_node_id` to support transparent proxies (EE) (RM12025, RM12033)
With CAMO enabled, BDR reports a `bdr.local_node_id` GUC to the client. To fully support transparent proxies like HAproxy, BDR now reports this value once per transaction in combination with `transaction_id`, to ensure a client doesn't ever return incorrect results from `PQparameterStatus()` because of a stale cache caused by missing a transparent connection switch.
- Fix global DDL and DML lock recovery after instance restart or crash (RM12042)
Previous versions of BDR might not correctly block the writes against global lock if the node or apply worker restarted after the lock was acquired. This could lead to divergent data changes in case the protected command(s) were changing data concurrently.
- Fix global DDL and DML lock blocking of replication changes (RM12042)
Previous versions of BDR would continue replication of changes to a locked table from other nodes. This could result in temporary replication errors or permanent divergent data changes if the transaction which acquired the global lock would be applied on some nodes with delay.
- Fix hang in cleanup/recovery of acquired global lock in the apply worker
The apply worker which acquired global lock for another node could on exit leak the hanging lock which could then get "stolen" by different backend. This could cause the apply worker to wait for lock acquisition of same

lock forever after restart.

- Don't hold back freezeLimit forever (EE) (RM11783)
The Enterprise Edition of BDR holds back freeze point to ensure enough info is available for conflict resolution at all times. Make sure that we don't hold the freeze past xid wraparound warning limit to avoid loss of availability. Allow the limit to move forward gracefully to avoid risk of vacuum freeze storms.
- Properly close connections in `bdr.run_on_all_nodes`
Removes log spam about connection reset by peer when `bdr.run_on_all_nodes` is used.
- Clarify docs that CREATE MATERIALIZED VIEW is not supported yet. (RT66363)

25.20 BDR 3.6.10

BDR 3.6.10 is the tenth minor release of the BDR 3.6 series. This release includes minor new features as well as fixes for issues identified previously.

25.20.1 Improvements

- Add new optional performance mode for CAMO - remote_write (EE) (RM6749)
This release enables a CAMO remote_write mode offering quicker feedback at time of reception of a pre-commit message from the CAMO partner, rather than only after the application of the transaction. Significantly better performance in exchange for small loss of robustness.
- Defer switching to CAMO mode until the partner has caught up (EE) (RM9605/RT65000/RT65827)
In async mode for improved availability, CAMO allows to switch to a local mode in case the CAMO partner is not reachable. When switching back, it may have to catchup before it can reasonably confirm new transactions from its origin. The origin now uses an estimate of the catchup time to defer the switch back to CAMO mode to eliminate transactions timing out due to the CAMO partner still catching up.
- Add functions `wait_for_apply_queue` and `wait_for_camo_partner_queue` (EE)
Allows to wait for transactions already received but currently queued for application. These can be used to prevent stale reads on a BDR node replicated to in remote_write mode.
- Improve monitoring of replication, especially around catchup estimates for peer nodes (EE) (RM9798)
Introduce two new views `bdr.node_replication_rates` and `bdr.node_estimates` to get a reasonable estimate of how far behind a peer node is in terms of applying WAL from this local node. The `bdr.node_replication_rates` view gives an overall picture of the outgoing replication activity in terms of the average apply rate whereas the `bdr.node_estimates` focuses on the catchup estimates for peer nodes.
- Support Column-Level Conflict Resolution for partitioned tables (EE) (RM10098, RM11310)
Make sure that newly created or attached partitions are setup for CLCD if their parent table has CLCD enabled.
- Take global DML lock in fewer cases (RM9609).
Don't globally lock relations created in current transaction, and also relations that are not tables (for example views) as those don't get data via replication.

25.20.2 Resolved Issues

- Disallow setting `external` storage parameter on columns that are part of a primary key (RM11336).
With such a setting, any UPDATE could not be replicated as the primary key would not get decoded by PostgreSQL.
- Prevent ABA issue with `check_full_tuple = true`. (RM10940, RM11233)
We only do the full row check if `bdr.inc_row_version()` trigger exists on a table from now on to prevent ABA issue when detecting conflict on UPDATES that didn't change any data when `check_full_tuple` is set to `true`.

25.21 BDR 3.6.9

BDR 3.6.9 is the ninth minor release of the BDR 3.6 series. This release includes minor new features as well as fixes for issues identified previously.

25.21.1 Improvements

- Parameters to help BDR assessment by tracking certain application events (EE)
`bdr.assess_update_replica_identity = IGNORE (default)| LOG| WARNING| ERROR` Updates of the Replica Identity (typically the Primary Key)
`bdr.assess_lock_statement = IGNORE (default)| LOG| WARNING| ERROR`
Two types of locks that can be tracked are: * `explicit table-level locking (LOCK TABLE ...)` by user sessions * `explicit row-level locking (SELECT ... FOR UPDATE/FOR SHARE)` by user sessions (RM10812, RM10813)

25.21.2 Resolved Issues

- Fix crash MIN/MAX for gsum and psum CRDT types (RM11049)
- Disallow parted nodes from requesting `bdr.part_node()` on other nodes. (RM10566, RT65591)

25.22 BDR 3.6.8

BDR 3.6.8 is the eighth minor release of the BDR 3.6 series. This release includes a fix for a critical data loss issue as well as fixes for other issues identified with previous releases.

25.22.1 Improvements

- Create the `bdr.triggers` view (EE) (RT65773) (RM10874) More information on the triggers related to the table name, the function that is using it, on what event is triggered and what's the trigger type.

25.22.2 Resolved Issues

- Loss of TOAST data on remote nodes replicating UPDATES (EE) (RM10820, RT65733)
A bug in the transform trigger code path has been identified to potentially set toasted columns (very long values for particular columns) to NULL when applying UPDATES on remote nodes, even when transform triggers have never been used. Only BDR-EE is affected and only when tables have a toast table defined and are not using REPLICA IDENTITY FULL. BDR3 SE is not affected by this issue. LiveCompare has been enhanced with damage assessment and data recovery features, with details provided in a separate Tech Alert to known affected users. This release prevents further data loss due to this issue.
- CAMO: Eliminate a race leading to inadvertent timeouts (EE) (RM10721)
A race condition led to pre-commit confirmations from a CAMO partner being ignored. This in turn caused inadvertent timeouts for CAMO-protected transactions and poor performance in combination with `synchronous_replication_availability` set to `async`. This fixes an issue introduced with release 3.6.7.
- CAMO: Properly handle transaction cancellation at COMMIT time (EE) (RM10741)
Allow the COMMIT of a CAMO-protected transaction to be aborted (more gracefully than via node restart or PANIC). Enable run-time reconciliation with the CAMO partner to make the CAMO pair eventually consistent.
- CAMO: Ensure the status query function keeps CAMO enabled. (EE) (RM10803)
The use of the `logical_transaction_status` function disabled CAMO for the entire session, rather than just for the query. Depending on how a CAMO client (or a proxy in between) used the session, this could lead to CAMO being inadvertently disabled. This has been fixed and CAMO remains enabled independent of calls of this function.
- Eager: cleanup stale transactions. (EE) (RM10595)
Ensures transactions aborted during their COMMIT phase are cleaned up eventually on all nodes.
- Correct TransactionId comparison when setting VACUUM freeze limit.
This could lead to ERROR: cannot freeze committed xmax for a short period at xid wrap, causing VACUUMs to fail. (EE) (RT65814, RT66211)

25.23 BDR 3.6.7.1

This is a hot-fix release on top of 3.6.7.

25.23.1 Resolved Issues

- Prevent bogus forwarding of transactions from a removed origin. (RT65671, RM10605) After the removal of an origin, filter transactions from that origin in the output plugin, rather than trying to forward them without origin information.

25.24 BDR 3.6.7

BDR 3.6.7 is the seventh minor release of the BDR 3.6 series. This release includes minor new features as well as fixes for issues identified previously.

25.24.1 Improvements

- CAMO and Eager switched to use two-phase commit (2PC) internally.
This is an internal change that made it possible to resolve a deadlock and two data divergence issues (see below). This is a node-local change affecting the transaction's origin node exclusively and has no effect on the network protocol between BDR nodes. BDR nodes running CAMO now require a configuration change to allow for enough `max_prepared_transactions`; see [Upgrading] for more details. Note that there is no restriction on the use of temporary tables, as exists in explicit 2PC in PostgreSQL.
- Add globally-allocated range sequences (RM2125)
New sequence kind which uses consensus between nodes to assign ranges of sequence numbers to individual nodes for each sequence as needed. Supports all of smallint, integer and bigint sequences (that includes serial column type).
- Implement Multi-Origin PITR Recovery (EE) (RM5826)
BDR will now allow PITR of all or some replication origins to a specific point in time, providing a fully consistent viewpoint across all subsets of nodes. 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. Thus we use physical WAL recovery using multiple separate logical stopping points for each origin. 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".
- Add `bdr.xact_replication` option for transaction replication control
Allows for skipping replication of whole transaction in a similar way to what `bdr.ddl_replication` does for DDL statements but it affects all changes including INSERT/UPDATE/DELETE. Can only be set via `SET LOCAL`. Use with care!
- Prevent accidental manual drop of replication slots created and managed by BDR
- Add `bdr.permit_unsafe_commands` option to override otherwise disallowed commands (RM10148)
Currently overrides the check for manual drop of BDR replication slot in the Enterprise Edition.
- Allow setting `bdr.ddl_replication` and `bdr.ddl_locking` as `bdr_superuser` using the SET command
This was previously possible only via the wrapper functions `bdr.set_ddl_replication()` and `bdr.set_ddl_locking()` which are still available.
- Improve performance of consensus messaging layer (RM10319, RT65396)

25.24.2 Resolved Issues

- Delete additional metadata in `bdr.drop_node` (RT65393, RM10346)
We used to keep some of the local node info behind which could prevent reuse of the node name.
- Correctly synchronize node-dependent metadata when using `bdr_init_physical` (RT65221, RM10409)
Synchronize additional replication sets and table membership in those as well as stream triggers and sequence information in `bdr_init_physical`, in a similar way to logical synchronization.
- Fix potential data divergence with CAMO due to network glitch (RM#10147)
This fixes an data inconsistency that could arise between the nodes of a CAMO pair in case of an unreachable or unresponsive (but operational) CAMO partner and a concurrent crash of the CAMO origin node. An in-flight COMMIT of a transaction protected by CAMO may have ended up getting committed on one node, but aborted on the other, after both nodes are operational and connected.
- Fix a potential deadlock between a cross-CAMO pair (RM#7907)
With two nodes configured as a symmetric CAMO pair, it was possible for the pair to deadlock, if both nodes were down and restarting, but both having CAMO transactions in-flight.

- Fix potential data divergence for Eager transaction in face of a crash (RM#9907)
In case of a crash of the origin node of an Eager transaction just before the final local commit, such a transaction could have ended up aborted on the origin but committed on all other nodes. This is fixed by using 2PC on the origin node as well and properly resolving in-flight Eager transaction after a restart of the origin node.
- Correct handling of fast shutdown with CAMO transactions in-flight (RM#9556)
A fast shutdown of Postgres on a BDR node that's in the process of committing a CAMO-protected transaction previously led to a PANIC. This is now handled gracefully, with the in-flight CAMO transaction still being properly recovered after a restart of that node.

25.25 BDR 3.6.6

BDR 3.6.6 is the sixth minor release of the BDR 3.6 series. This release includes minor new features as well as fixes for issues identified previously.

25.25.1 Improvements

- Add `bdr.drop_node()` (RM9938)
For removing node metadata from local database, allowing reuse of the node name in the cluster.
- Include `bdr_init_physical` in BDR-SE (RM9892)
Improves performance during large node joins - BDR-EE has included this tool for some time.
- Enhance `bdr_init_physical` utility in BDR-EE (RM9988) Modify `bdr_init_physical` to optionally use selective `pg_basebackup` of only the target database as opposed to the earlier behavior of backup of the entire database cluster. Should make this activity complete faster and also allow it to use less space due to the exclusion of unwanted databases.
- TRUNCATE is now allowed during eager replicated transactions (RM9812)
- New `bdr.global_lock_table()` function (RM9735).
Allows explicit acquire of global DML lock on a relation. Especially useful for avoidance of conflicts when using TRUNCATE with concurrent write transactions.
- New conflict type `update_pkey_exists` (RM9976)
Allows conflict resolution when a PRIMARY KEY was updated to one which already exists on the node which is applying the change.
- Reword DDL locking skip messages and reduce log level
The previous behavior was too intrusive.
- Add `bdr.apply_log_summary` (RM6596)
View over `bdr.apply_log` which shows the human-readable conflict type and resolver string instead of internal id.
- Add `bdr.maximum_clock_skew` and `bdr.maximum_clock_skew_action` configuration options (RM9379)
For checking clock skew between nodes and either warning or delaying apply in case the clock skew is too high.

25.25.2 Resolved Issues

- Move CRDT type operators from public schema to `pg_catalog` (RT65280, RM10027)
Previously BDR operators were installed in public schema, preventing their use by servers implementing stricter security policies. No action required.
- Remember if unsupported Eager Replication command was run in current transaction.
This allows us to prevent situations where an unsupported command was run while Eager Replication was turned off and later in the transaction the Eager Replication is turned on.
- Fix the “!” operator for `crdt_pnsum` data type (RM10156)
It’s the operator for resetting the value of the column, but in previous versions the reset operation didn’t work on this type.

25.26 BDR 3.6.5

BDR 3.6.5 is the fifth minor release of the BDR 3.6 series. This release includes minor new features as well as fixes for issues identified in 3.6.4.

25.26.1 Improvements

- Allow late enabling of CAMO (RM8886)
The setting `bdr.enable_camo` may now be turned on at any point in time before a commit, even if the transaction already has an id assigned.
- Add version-2 KSUUIDs which can be compared using simple comparison operators (RM9662)
- New `delete_recently_updated` conflict type (RM9673/RT65063) Triggered by DELETE operation arriving out of order - the DELETE has an older commit timestamp than the most recent local UPDATE of the row. Can be used to override the default policy of DELETE always winning.
- Make bdr admin function replication obey DDL replication filters (RT65174)
So that commands like `bdr.replication_set_add_table` don’t get replicated to a node which didn’t replicate CREATE TABLE in the first place.
- Don’t require group replication set to be always subscribed by every node (RT65161)
Since we now apply DDL replication filters to admin functions, it’s no longer necessary to force group replication set to be subscribed by every node as other replication sets can be configured to replicate the admin function calls.
- Allow a few more DDL operations to skip the global DML lock
The following DDL operations have been optimized to acquire only a global DDL lock, but not the DML one:

- ALTER TABLE .. ALTER COLUMN .. SET STATISTICS

 - ALTER TABLE .. VALIDATE CONSTRAINT
 - * ALTER TABLE .. CLUSTER ON
 - * ALTER TABLE .. RESET

 - CREATE TRIGGER
- Add new BDR trigger that resolves Foreign Key anomalies on DELETE (RM9580)
- Add new function `bdr.run_on_all_nodes()` to assist monitoring and diagnostics (RM9945)

- Extend the CAMO reference client in C
Allow setting a `bdr.commit_scope` for test transactions.
- Prevent replication of CLUSTER command to avoid operational impact
- To assist with security and general diagnostics, any DDL that skips replication or global DDL locking at user request will be logged. For regular users of non-replicated and/or non-logged DDL this may increase log volumes. Some log messages have changed in format. This change comes into effect when `bdr.ddl_locking = off` and/or `bdr.ddl_replication = off`.
- Greatly enhance descriptions of BDR admin functions with regard to (RM8345)
their operational impact, replication, locking and transactional nature
- Detailed docs to explain concurrent Primary Key UPDATE scenarios (RM9873/RT65156)
- Document examples of using `bdr.replication_set_add_ddl_filter()` (RM9691)

25.26.2 Resolved Issues

- Rework replication of replication set definition (RT64451) Solves the issue with the replication set disappearing from some nodes that could happen in certain situations.
- Acquire a Global DML lock for these DDL commands for correctness (RM9650)
 - CREATE UNIQUE INDEX CONCURRENTLY
 - DROP INDEX CONCURRENTLY
 - `bdr.drop_trigger()` admin function since adding or removing any constraint could allow replication-halting DML
- Correctly ignore nodes that are parting or parted in the Raft code (RM9666/RT64891) Removes the excessive logging of node membership changes.
- Don't try to do CRDT/CLCD merge on `update_recently_deleted` (RM9674) It's strictly row-level conflict; doing a merge would produce the wrong results.
- Allow BDR apps that require `standard_conforming_strings = off` (RM9573/RT64949)
- Use replication slot metadata to postpone freezing of rows (RM9670) (EE-only) Otherwise an `update_origin_change` conflict might get undetected after a period of node downtime or disconnect. The SE version can only avoid this using parameters.
- Correct `bdr_wait_slot_confirm_lsn()` to wait for the LSN of last commit, rather than the LSN of the current write position. In some cases that could have released the wait earlier than appropriate, and in other cases it might have been delayed.

25.27 BDR 3.6.4

BDR 3.6.4 is the fourth minor release of the BDR 3.6 series. This release includes minor new features as well as fixes for issues identified in 3.6.3.

25.27.1 The Highlights of BDR 3.6.4

- Apply statistics tracking (RM9063) We now track statistics about replication and resource use for individual subscriptions and relations and make them available in the `pglogical.stat_subscription` and `pglogical.stat_relation` views. The tracking can be configured via the `pglogical.stat_track_subscription` and `pglogical.stat_track_relation` configuration parameters.
- Support CAMO client protocol with Eager All Node Replication Extend `bdr.logical_transaction_status` to be able to query the status of transactions replicated in global commit scope (Eager All Node Replication). Add support for Eager All Node Replication in the Java CAMO Reference client.

25.27.2 Resolved Issues

- Fix initial data copy of multi-level partitioned tables (RT64809) The initial data copy used to support only single level partitioning; multiple levels of partitioning are now supported.
- Don't try to copy initial data twice for partitions in some situations (RT64809) The initial data copy used to try to copy data from all tables that are in replication sets without proper regard to partitioning. This could result in partition data being copied twice if both the root partition and individual partitions were published via the replication set. This is now solved; we only do the initial copy on the root partition if it's published.
- Fix handling of indexes when replicating INSERT to a partition (RT64809) Close the indexes correctly in all situations.
- Improve partitioning test coverage (RM9311) In light of the partitioning related issues, increase the amount of automated testing done against partitioned tables.
- Fix merging of `crdt_pnsum` data type (RT64975) The internal index was handled wrongly, potentially causing a segmentation fault; this is now resolved.
- Fix cache lookup failed on database without BDR extension installed (RM9217) This could previously lead to errors when dropping tables on a PostgreSQL instance which has the BDR library loaded but does not have the extension installed.
- Fix permission issues on `bdr.subscription_summary` (RT64945) No need to have permissions on `pglogical.get_sub_progress_timestamp()` to use this view anymore.
- Cleanup prepared Eager All Node transactions after failures (RM8996) Prevents inconsistencies and hangs due to unfinished transactions after node or network failures. Uses Raft to ensure consistency between the nodes for the cleanup of such dangling prepared transactions.

25.27.3 Other Improvements

- The `replicate_inserts` option now affects initial COPY We now do initial copy of data only if the table replicates inserts.
- Lower log level for internal node management inside Raft worker (RT64891) This was needlessly spamming logs during node join or parting.
- Warn when executing DDL without DDL replication or without DDL locking The DDL commands executed without DDL replication or locking can lead to divergent databases and cause replication errors so it's prudent to warn about them.
- Allow create statistics without dml lock (RM9507)
- Change documentation to reflect the correct default settings for the `update_missing` conflict type.

25.28 BDR 3.6.3

BDR 3.6.3 is the third minor release of the BDR 3.6 series. This release includes minor new features as well as fixes for issues identified in 3.6.2.

25.28.1 The Highlights of BDR 3.6.3

- Add btree/hash operator classes for CRDT types (EE, RT64319) This allows the building of indexes on CRDT columns (using the scalar value) and the querying of them using simple equality/inequality clauses, using the in GROUP BY clauses etc.
- Add implicit casts from int4/int8 for CRDT sum types (EE, RT64600)
To allow input using expressions with integer and CRDT sum types together. For example:

```
CREATE TABLE t (c bdr.crdt_gsum NOT NULL DEFAULT 0);
```

- New `update_recently_deleted` conflict type (RM8574)
Conflicts are handled differently for the special case of `update_missing` when BDR detects that the row being updated is missing because it was just recently deleted. See UPDATE/DELETE Conflicts in the documentation for details.
- Allow DDL operations in CAMO protected transactions, making automatic disabling of CAMO obsolete (EE, RT64769)
- Add the connection status checking function `bdr.is_camo_partner_connected` for CAMO (EE). See the Commit At Most Once documentation for details.
- Persist the `last_xact_replay_timestamp` (RT63881)
So that it's visible even if the subscription connection is down (or remote node is down).
- Major documentation improvements
Copy-edit sentences to make more sense, add extra clarifying info where the original wording was confusing.

25.28.2 Resolved Issues

- Use group locking for global DML lock (RT64404) This allows better cooperation between the global DML locker and the writers which are doing catch up of the remaining changes.

25.28.3 Other Improvements

- Support mixed use of legacy CRDT types and new CRDT types which are in bdr schema Implicitly cast between the two so their mixed usage and potential migration is transparent.
- Improve static code scanning Every build is scanned both by Coverity and Clang scan-build.
- Log changes of `bdr.ddl_replication` and `bdr.ddl_locking` Helps with troubleshooting when divergent DDL was run.
- Rework documentation build procedure for better consistency between HTML and PDF documentation. This mainly changes the way docs are structured into chapters so that there is a single source of chapter list and ordering for both PDF and HTML docs.

25.29 BDR 3.6.2

BDR 3.6.2 is the second minor release of the BDR 3.6 series. This release includes minor new features as well as fixes for issues identified in 3.6.1

25.29.1 The Highlights of BDR 3.6.2

- All the SQL visible interfaces are now moved to the `bdr` schema (EE) The CRDT types and per column conflict resolution interfaces are now in the `bdr` schema instead of `bdr_crdt` and `bdr_conflicts`. The types and public interfaces still exist in those schemas for compatibility with existing installations, however their use is not recommended as they are now deprecated and may be removed in a future release. Please use the ones in `bdr` schema. Documentation only contains references to the `bdr` schema now as well.
- Add `bdr.node_conflict_resolvers` view (RT64388) Shows current conflict resolver setting for each conflict type on the local node including the defaults.
- Add a CAMO Reference Client implementation in C and Java to the documentation.
- Support DEFERRED UNIQUE indexes They used to work only in limited cases before this release.

25.29.2 Resolved Issues

- Fix consensus request timeout during leader election (RT64569) The timeout wasn't applied when the leader was unknown leading to immediate failures of any action requiring consensus (for example global DDL locking). This is now resolved.
- Improve cleanup on failure during a DDL locked operation, This speeds up DDL locking subsystem recovery after error so that errors don't create a cascading effect.
- Unify the replication of admin function commands (RT64544) This makes the replication and locking behavior of administration function commands more in-line with DDL in all situations, including logical standby.
- Support covering UNIQUE indexes (RT64650) Previously, the covering UNIQUE indexes could result in ambiguous error messages in some cases.
- Switch to monotonic time source for Raft timing (RM6390) This improves reliability of Raft internal timing in presence of time jumps caused by NTPd and similar. As a result Raft reliability is improved in general.
- Improve locking in the internal connection pooler For more reliable messaging between nodes.
- Restore consensus protocol version on restart (RT64526) This removes the need for renegotiation every time a consensus worker or a node is restarted, making the features depending on newer protocol version consistently available across restarts.
- Correct automatic disabling and re-enabling of `bdr.enable_camo` when using DDL in a transaction. Ensure it cannot be manually re-enabled within the same transaction.
- Fix handling of CAMO confirmations arriving early, before the origin starts to wait. This prevents timeouts due to such a confirmation being ignored.

25.30 BDR 3.6.1

BDR 3.6.1 is the first minor release of the BDR 3.6 series. This release includes minor new features and fixes including all the fixes from 3.6.0.1 and 3.6.0.2.

25.30.1 The highlights of 3.6.1

- Add `bdr.role_replication` configuration option (RT64330) The new option controls the replication of role management statements (CREATE/ALTER/DROP/GRANT ROLE). This option is dependent on `bdr.ddl_replication` as the role management statements still obey the standard rules of the DDL replication. By default this is set to `on`, meaning that these statements are replicated if executed in a BDR-enabled database.
- Add `--standby` option to `bdr_init_physical` (RM8543, EE) Allows the creation of a logical standby using `bdr_init_physical`; previously only a full blown send/receive node could be created this way.
- Add `last_xact_replay_timestamp` to `bdr.subscription_summary` (RT63881) Shows the commit timestamp of the last replayed transaction by the subscription.
- Stop join on unrecoverable error (RT64463) Join might fail during the structure synchronization, which currently is an unrecoverable error. Instead of retrying like for other (transient) errors, just part the joining node and inform the user that there was an error.

25.30.2 Resolved Issues

- Improve the trigger security checking (RT64412) Allow triggers to have a different owner than the table if the trigger uses `bdr` or `pglogical` trigger functions, security definer functions (as those redefine security anyway) and also always allow replication set membership changes during initial replication set synchronization during the node join.
- Make BDR replicated commands obey `bdr.ddl_replication` (RT64479) Some of the BDR function calls (like `bdr_conflicts.column_timestamps_enable`) are replicated in a similar way as normal DDL commands including the DDL locking as appropriate. These commands in previous versions of BDR however ignored the `bdr.ddl_replication` setting and were always replicated. This is now fixed. In addition just like normal DDL, these commands are now never replicated from the logical standby.
- Don't try to replicate generic commands on global objects Several commands on global objects would be replicated even in situations where they shouldn't be because of how they are represented internally. Handling of the following commands has been fixed:
 - `ALTER ROLE/DATABASE/TABLESPACE ... RENAME TO`
 - `ALTER DATABASE/TABLESPACE ... OWNER TO`
 - `COMMENT ON ROLE/DATABASE/TABLESPACE`
 - `SECURITY LABEL ON ROLE/DATABASE/TABLESPACE`
- Properly timeout on CAMO partner and switch to Local mode (RT64390, EE) Disregard the connection status of other BDR nodes and switch to Local mode as soon as the designated CAMO partner node fails. Makes the switch to Local mode work in a four or more node cluster.

25.31 BDR 3.6.0.2

The BDR 3.6.0.2 release is the second bug-fix release in the BDR 3.6 series.

25.31.1 Resolved Issues

- Dynamic disabling of CAMO upon the first DDL (EE, RT64403)
- Fix hang in node join caused by timing issues when restoring Raft snapshot (RT64433)
- Fix the trigger function ownership checks (RT64412)
- Improve behavior of `promote_node` and `join_node_group` with `wait_for_completion := false`

25.32 BDR 3.6.0.1

The BDR 3.6.0.1 is the first bug-fix release in the BDR 3.6 series.

25.32.1 Resolved Issues

- Support `target_table_missing` conflict for transparent partitioning (EE) (RT64389)
- Fix message broker sometimes discarding messages (common side-effect are DDL locking timeouts)
- Raft protocol negotiations improvements
- Fixed memory leak in tracing code
- Improve synchronous `remote_write` replication performance (RT64397)
- Fixed commit timestamp variant handling of CLCD (EE)
- Re-add support for binary protocol
- Correct Local mode for CAMO with `synchronous_replication_availability = 'async'` (EE)
- Disallow and provide a hint for unsupported operations in combination with CAMO (EE).
- Fix deadlock in `logical_transaction_status` (EE)

25.33 BDR 3.6.0

The version 3.6 of BDR3 is a major update which brings improved CAMO, performance improvements, better conflict handling and bug fixes.

25.33.1 The highlights of BDR 3.6

- Differentiate BDR RemoteWrite mode and set `write_lsn`
- Significant replication performance improvement
 - Cache table synchronization state
 - Only send keepalives when necessary
 - Only do flush when necessary
 - Serialize transactions in fewer cases in wal sender (2ndQPostgres)
- Improved replication position reporting which is more in line with how physical streaming replication reports it
- Conflict detection and resolution improvements
 - Add new types of conflicts (like `target_table_missing`)
 - Add new types of conflict resolvers
 - Make conflict resolution configurable per node and conflict type
 - Improve conflict detection for updates
- Simplification of CAMO configuration (EE)
- Performance improvements for CAMO (EE)

25.33.2 Resolved issues

- Fix reporting of replay lag (RT63866)
- Fix CRDTs and conflict triggers for repeated UPDATES of same row in transaction (RT64297)
- Don't try to replicate REINDEX of temporary indexes

25.33.3 Other improvements

- Improved vacuum handling of Raft tables
- Improve and clarify CAMO documentation (EE)

'APPENDIX B

Appendix B: Conflict Details

'APPENDIX C

Appendix C: Known Issues

'APPENDIX D

Appendix D: Libraries

Appendix E: CAMO Reference Clients