LiveCompare

Version 2

Built at 2024-05-17T14:44:06

1 LiveCompare

LiveCompare is designed to compare any number of databases to verify they're identical. The tool compares the databases and generates a comparison report, a list of differences, and handy DML scripts so you can optionally apply the DML and fix the inconsistencies in any of the databases.

By default, the comparison set includes all tables in the database. LiveCompare allows checking of multiple tables concurrently (multiple worker processes) and is highly configurable to allow checking just a few tables or just a section of rows in a table.

Each database comparison is called a *comparison session*. When the program starts for the first time, it starts a new session and starts comparing table by table. In standalone mode, once all tables are compared, the program stops and generates all reports. You can start and stop LiveCompare without losing context information, so you can run it at convenient times.

Each table comparison operation is called a *comparison round*. If the table is too big, LiveCompare splits the table into multiple comparison rounds that are also executed in parallel, alongside other tables that are being carried on by other workers at the same time.

In standalone mode, the initial comparison round for a table starts from the beginning of the table (oldest existing PK) to the end of the table (newest existing PK). New rows inserted after the round starts are ignored. LiveCompare sorts the PK columns to get min and max PK from each table. For each PK column that's unsortable, LiveCompare casts its content to `string`. In PostgreSQL, you achieve this by using `::text`. In Oracle, use `to_char`.

When executing the comparison algorithm, each worker requires N+1 database connections, where N is the number of databases being compared. The extra required connection is to an output/reporting database, where the program cache is kept too, enabling you to stop and resume a comparison session.

You can manually recheck any differences found by the comparison algorithm at a later, convenient time. We recommend doing this to allow a replication consistency check. Upon the difference recheck, replication might have caught up on that specific row and the difference doesn't exist anymore, so the difference is removed. Otherwise it's marked as permanent.

At the end of the execution, the program generates a DML script so you can review it and fix differences one by one. Or you can apply the entire DML script to fix all permanent differences.

You can potentially use LiveCompare to ensure logical data integrity at the row level, for example, for these scenarios:

- Database technology migration (Oracle x Postgres).
- Server migration or upgrade (old server x new server).
- Physical replication (primary x standby).
- After failover incidents, for example to compare the new primary data against the old, isolated primary data.
- In case of an unexpected split-brain situation after a failover. If the old primary wasn't properly fenced and the application wrote data into it, you can use LiveCompare to know exactly the data that's present in the old primary and isn't present in the new primary. If they want, the DBA can use the DML script that LiveCompare generates to apply those data into the new primary.
- Logical replication. Three kinds of logical replication technologies are supported: Postgres native logical replication, pglogical, and EDB Postgres Distributed (PGD, formerly known as BDR).

Comparison performance

LiveCompare is optimized for use on production systems and has various parameters for tuning. Comparison rounds are read-only workloads. An example use case compared 43,109,165 rows in 6 tables in 9m 17s with 4 connections and 4 workers, giving comparison performance of approximately 77k rows per second, or 1 billion rows in <4 hours.

This use case is a general use case. For low-load, testing, migration, and other specific scenarios, you might be able to improve speed by changing the `data_fetch_mode` setting to use server-side cursors. In our experiments, each kind of server-side cursors provides an increase in performance on use cases involving either small or large tables.

Security considerations for the user

For PostgreSQL 13 and earlier, LiveCompare requires a user that can read all data being compared. PostgreSQL 14 introduced a new role, pg_read_all_data, that can be used for LiveCompare.

When `logical_replication_mode = bdr`, LiveCompare requires a user with the bdr_superuser role. When `logical_replication_mode = pglogical`, LiveCompare requires a user with the `pglogical_superuser` role.

To apply the DML scripts in PGD, all divergent connections (potentially all data connections) require a user with the bdr_superuser role to disable `bdr.xact_replication`.

If PGD is being used, LiveCompare associates all fixed rows with a replication origin called `bdr_local_only_origin`. LiveCompare also applies the DML with the transaction datetime far in the past, so if there are any PGD conflicts with real DML being executed on the database, LiveCompare DML always loses the conflict.

With the default setting of `difference_fix_start_query`, the transaction in apply scripts changes role to the owner of the table to prevent database users from gaining access to the role applying fixes by writing malicious triggers. As a result, the user for the divergent connection needs to be able to switch role to the table owner.

2 Release notes

The LiveCompare documentation describes the latest version of LiveCompare 2 including minor releases and patches. The release notes in this section provide information on what is new in each release.

Version	Release Date
2.5.1	May 16 2024
2.5	May 09 2023
2.4	Nov 29 2022
2.3	Aug 15 2022
2.2	Jun 14 2022
2.1	Mar 31 2022
2.0	Feb 15 2022

2.1 LiveCompare 2.5.1 release notes

Released: 16 May 2024

LiveCompare 2.5.1 includes the following new features, enhancements, bug fixes, and other changes:

Type	Description	Support Ticket
Platform Support	LiveCompare is now supported for use on x86 based Debian 12 platforms.	
Bug Fix	Fixed an issue where LiveCompare generates DML with 'None' as the setting for bytea columns instead of null.	93247/25921

Type	Description	Support Ticket
Bug Fix	Restored the ability to connect to Oracle 10g databases and enforcing `comparison_algorithm = full_row` be set for such connections. Note that EDB no longer tests LiveCompare with Oracle 10g. As such, it is no longer on the list of Oracle versions officially supported with LiveCompare. Oracle 10g had previously been supported and is expected to continue to work in most cases. However, some limitations may exist.	99815/31762
Bug Fix	Fixed the issue caused by the differences in how the primary keys returned by the EPAS queries and Oracle queries were ordered.	
Bug Fix	Fixed an issue where certain column names were incorrectly being identified as Oracle reserved words.	102498/34229

2.2 LiveCompare 2.5 release notes

Released: 09 May 2023

LiveCompare 2.5 includes the following new features, enhancements, bug fixes, and other changes:

Type	Description
Feature	Support for EDB Postgres Distributed 5.
Enhancement	LiveCompare now holds a version agnostic list of reserved words for Oracle and Postgres, used to properly handle quoted identifiers.
Bug fix	Fixed an issue where the HandleIgnoredColumns method was failing in mixed comparison mode.
Bug fix	Fixed an issue where the PrepareMask method was failing when using a custom collate.
Bug fix	Fixed an issue where using the `node_name` caused LiveCompare to try the pglogical metadata for PGD 4 and 5.

2.3 LiveCompare 2.4 release notes

Released: 29 Nov 2022

LiveCompare 2.4 includes the following new features, enhancements, bug fixes, and other changes:

Type	Description
Bug fix	Fixed an issue where QueryTablePKColumns would run slowly on tables with composite primary keys.
Bug fix	Fixed an issue where LiveCompare was not handling time columns correctly.
Bug fix	Fixed an issue where connection strings containing single-quotes or apostrophes were not handled correctly.
Bug fix	Fixed issue whereby the `--recheck` option could throw an unhandled exception in some circumstances.

2.4 LiveCompare 2.3 release notes

Released: 15 Aug 2022

LiveCompare 2.3 includes the following new features, enhancements, bug fixes, and other changes:

Type	Description	ID
Enhancement	Support for RedHat Enterprise Linux (RHEL) 8 on IBM Power (ppc64le).	
Enhancement	Support for SLES 12 SP5 on IBM Power (ppc64le).	
Enhancement	Support for SLES 15 SP3 on IBM Power (ppc64le).	
Enhancement	Support for Ubuntu 22.04 (Jammy) on x86_64.	
Enhancement	Support for Debian 11 (Bullseye) on x86_64.	

2.5 LiveCompare 2.2 release notes

Released: 14 Jun 2022

LiveCompare 2.2 includes the following new features, enhancements, and bug fixes:

Type	Description	ID
Feature	Added `--dry-run` execution mode, which allows users to validate the `.ini` file and view some useful comparison information, without running comparison session. See Dry-run mode for more information.	LIV-142, RT78462
Feature	Any abort messages received during the comparison session are printed in chronological order.	
Feature	Displays the list of connections, including technology, version, if the connection is a tiebreaker or a source of truth, and if it's reachable.	
Feature	Displays the `Table Filter` if it is configured.	
Feature	Lists the tables that are included in the comparison. This is the list of common tables that exist across all connections, after applying the `Table Filter`. For each table, shows the `Column Filter`, `Row Filter` and `Comparison Key`, if applicable.	
Enhancement	Support for SLES 12 on x86_64.	LIV-112
Enhancement	Support for SLES 15 on x86_64.	LIV-113
Enhancement	Updated the list of reserved words according to Postgres `kwlist` source code.	LIV-141, RT80745
Enhancement	The main progress bar percentage is now using a float instead of an integer. Previously, the number was rounded up.	LIV-138, RT80745
Enhancement	When `edb_redwood_date` is enabled in EPAS, the `date` columns are created as `timestamp` columns. This type mismatch was previously detected as a mismatch in the Common Hash, which triggers a full row comparison. Performance is improved by handling this mismatch in the Common Hash (which is faster than a full row), by checking the `edb_redwood_date` setting in these cases.	LIV-153
Enhancement	Demoted hash mismatch log messages from WARNING to DEBUG.	LIV-145
Enhancement	Logs now include the `application_name` in the message.	LIV-141

Type	Description	ID
Enhancement	Documented the behavior of using the current timestamp in `Row Filter` on Postgres or EPAS when `data_fetch_mode = prepared_statements` (the default). In this situation, it's also required to set `data_fetch_mode = server_side_cursors_with_hold` or `server_side_cursors_without_hold`.	LIV-155
Enhancement	Improved performance when generating the apply DML scripts when there is an increased number of divergences detected.	LIV-159
Bug fix	Fixed a problem where an array column being considered as a Comparison Key caused the comparison on a table to be aborted due to an exception.	LIV-38, LIV-154, RT81758
Bug fix	A problem was resolved where the number of divergent and processed rows was not being properly updated after the comparison round, only by a Heart Beat. In this case, the rows were outdated, showing only the position since the last Heart Beat. For tables where the comparison took less than `min_time_between_heart_beats`, it would always indicate zero.	LIV-149, RT75805
Bug fix	Fixed a corner case where unhandled exceptions could cause the comparison worker to hang.	LIV-140, RT80745
Bug fix	Normalizing decimal values to avoid false positives when comparing Oracle versus Postgres.	LIV-153
Bug fix	Fix a corner case in which if `comparison_algorithm = block_hash` and `buffer_size = 1`, and there were any divergences found, the comparison would not advance due to an issue in the cursor advancing algorithm.	LIV-150
Bug fix	Allow the same table to be configured in multiple filter sections.	LIV-156

2.6 LiveCompare 2.1 release notes

Released: Mar 31 2022

LiveCompare 2.1 includes the following new features, bug fixes, and other changes:

Type	Description	ID
Feature	Support for EDB Postgres Distributed 4.	LIV-131
Feature	New setting `min_time_between_heart_beats`, which tells LiveCompare to log the comparison progress at every heart beat, by default set to 30 seconds using the `INFO` log level.	LIV-128
Feature	New settings `comparison_cost_limit` and `comparison_cost_delay` that, when greater than 0, tell each worker to take a nap of `comparison_cost_delay` seconds (for example, `0.5`) after processing `comparison_cost_limit` number of rows.	LIV-16
Change	Default value for `parallel_chunk_rows` set to `0`, which disables table splitting by default, as recent investigation proved to cause performance decrease for general use cases. For more information, see Compare mode.	LIV-130

Type	Description	ID
Change	Demoted to `DEBUG` the log message about the number of processed rows from `CanAdvanceCursors` method.	LIV-129
Bug fix	Fixed an issue for Oracle versus Postgres comparisons of the `timestamp(6)` data type where failing with `ORA-01830`.	LIV-127

2.7 LiveCompare 2.0 release notes

Released: 15 Feb 2022

LiveCompare 2.0 includes the following new features, enhancements, bug fixes, and other changes:

Type	Description	ID
Feature	New section in setting called `Comparison Key` allows users to define a custom comparison key (list of columns) per table. This can be useful for tables without primary keys or unique indexes. See Comparison Key for more information.	LIV-5
Feature	If the table has no user-defined comparison key or primary key, LiveCompare now tries to use the unique indexes from the table. As tables can have multiple unique indexes, LiveCompare prefers to use the index where columns are not nullable. If not possible, then LiveCompare tries to use the first unique index that has less columns. If there is no unique indexes, then LiveCompare tries to use all columns from the table as a comparison key. Please note that LiveCompare does not try to ignore nullable columns from unique indexes.	LIV-39
Feature	When using all columns from the table as a comparison key, LiveCompare now ignores the nullable columns to avoid false positives when considering all columns. This behavior can be disabled by setting `ignore_nullable = false`.	LIV-96
Feature	When using an user-defined comparison key or all columns from the table as a comparison key, LiveCompare now checks if they would enforce uniqueness. If when using the column set there is any duplicate values, then LiveCompare aborts the comparison on the table. This behavior can be disabled by setting `check_uniqueness_enforcement = false`.	LIV-119
Enhancement	Added support to Oracle 21c.	LIV-47
Enhancement	On Oracle 12c and newer, LiveCompare is already able to use a common hash to allow `comparison_algorithm = block_hash` and `comparison_algorithm = row_hash`, which are faster and allow table splitting among multiple worker processes. This is done on Oracle side using the `standard_hash()` function, which was introduced on Oracle 12c. Now LiveCompare also allows `comparison_algorithm = block_hash` and `comparison_algorithm = row_hash` on Oracle 11g, by using the equivalent function `sys.dbms_crypto.hash()`, provided that the user has `EXECUTE` privileges on the `sys.dbms_crypto` Oracle system package.	LIV-79

Type	Description	ID
Enhancement	LiveCompare schema can now be added to a replication-enabled (EDB Postgres Distributed, pglogical or native logical replication) database.	LIV-42
Enhancement	LiveCompare can use the new `pg_read_all_data` role in PostgreSQL 14.	LIV-73
Enhancement	Abort with a proper message if any database version is not supported.	
Change	Package has been renamed to `edb-livecompare` from `2ndq-livecompare`.	
Change	Executable has been renamed to `livecompare` from `2ndq-livecompare`.	
Bug fix	Properly quote the unicode sequence `\u0000` to avoid an error when generating DML.	LIV-98
Bug fix	Fixed an issue where the number of total rows was displayed incorrectly when the table was split into multiple round parts.	LIV-14
Bug fix	Fixed an issue where empty `BLOB` on Oracle when compared against an empty `bytea` on Postgres was generating a false positive.	LIV-103
Bug fix	Fixed an issue where connectivity issues were causing exceptions aborting the whole comparison session. Now LiveCompare is able to reconnect and continue the comparison if possible.	LIV-84
Bug fix	Fixed an unhandled exception on the recheck mode if there are any divergences.	LIV-107
Bug fix	Fixed an issue where the table comparison was not being aborted if the table didn't exist on a connection and `logical_replication_mode` was disabled.	LIV-108
Bug fix	Fixed an issue where fields of `timestamp` data type were always generating a mismatching hash between Oracle and Postgres.	
Bug fix	Fixed an issue where ignored columns were still being considered in the common hash.	

3 Requirements

LiveCompare requires:

- Python 3.6 or 3.7
- PostgreSQL / EDB Postgres Extended 9.5+ / EDB Postgres Advanced Server 11+ (on the output connection)
- PostgreSQL / EDB Postgres Extended 9.4+ / EDB Postgres Advanced Server 11+ or Oracle 11g+ (on the data connections being compared)

LiveCompare requires Debian 10+, Ubuntu 16.04+, SLES 12 SP5 and 15 SP3, or CentOS/RHEL/RockyLinux/AlmaLinux 7+.

You can install LiveCompare from the EnterpriseDB `products/livecompare` repository. For details, see the EDB customer portal.

LiveCompare installs on top of either:

- The latest Python version for Ubuntu, Debian, and CentOS/RHEL 8, as provided by the `python3` packages
- Python 3.6 for CentOS/RHEL 7, as provided by the `python-36` packages

On CentOS/RHEL distributions, LiveCompare also requires the EPEL repository. For details, see the EPEL webpage.

Specifically on CentOS/RHEL version 7, the Python component `tqdm` is too old (< 4.16.0). You can install the latest `tqdm` using `pip` or `pip3` for the user that is running LiveCompare:

```
pip install --user tqdm --upgrade
```

If running LiveCompare against an Oracle database, Oracle Instant Client must be installed. See Oracle support requirements for more information.

LiveCompare with TPAexec

You can use the following sample config for `TPAexec` to build a server with `LiveCompare` and `PostgreSQL 11`:

```
---
architecture: M1
cluster_name:
livecompare_m1
cluster_tags: {}

cluster_vars:
  postgres_coredump_filter: '0xff'
  postgres_version: '13'
  postgresql_flavour: postgresql
  repmgr_failover:
manual
  tpa_2q_repositories:
  - products/livecompare/release
  packages:
    common:
    - edb-livecompare
  use_volatile_subscriptions: true

locations:
- Name: main

instance_defaults:
```

```
  image: tpa/rocky
  platform:
docker
  vars:
    ansible_user: root

instances:
- Name:
livem1node1
  location: main
  node: 1
  role: primary
  published_ports:
    - 5401:5432
- Name:
livem1node2
  location: main
  node: 2
  role: replica
  upstream:
livem1node1
  published_ports:
    - 5402:5432
```

For details about TPAexec, see the EDB customer portal.

4 Supported technologies

LiveCompare can connect to and compare data from a list of technologies, including PostgreSQL, EDB Postgres Distributed (PGD, formerly known as BDR), and Oracle.

LiveCompare has three kinds of connections:

- **Initial** (optional): Used to fetch metadata about pglogical or PGD connections. Required if data connections are pqloqical or PGD, and if `replication_sets` or `node_name` settings are used. Requires `logical_replication_mode = pglogical` or `logical_replication_mode = bdr` . A pglogical- or PGD-enabled database is required.
- **Data**: The actual database connection that the tool connects to to perform data comparison. The first connection in the list is used to solve `Table Filter` and `Row Filter` , and is also used with the `Initial Connection' to gather information about PGD nodes. If` logical_replication_mode = bdr `and` all_bdr_nodes = on `, then LiveCompare considers all PGD nodes that are part of the same PGD cluster as the` Initial Connection`. In this case, you don't need to define data connections individually. The fix can be potentially applied in all data connections, as comparison and consensus decisions work per row.
- **Output** (mandatory): Where LiveCompare creates a schema called `livecompare` , some tables, and views. This is required to keep progress and reporting data about comparison sessions. It must be a PostgreSQL or 2ndQPostgres connection.

The table shows versions and details about supported technologies and the context in which you can use them in LiveCompare.

Technology	Versions	Possible connections
PostgreSQL	10, 11, 12, 13, 14, 15, and 16	Data and output
EDB PostgreSQL Extended (PGE)	10, 11, 12, 13, 14, 15, and 16	Data and output
EDB PostgreSQL Advanced Server (EPAS)	11, 12, 13, 14, 15, and 16	Data and output
pglogical	2 and 3	Initial, data, and output
EDB Postgres Distributed (PGD)	1, 2, 3, 4, and 5	Initial, data, and output
Oracle	11g, 12c, 18c, 19c, and 21c	A single data connection

Note that EDB no longer tests LiveCompare with Oracle 10g. As such, it is no longer on the list of Oracle versions officially supported with LiveCompare. Oracle 10g had previously been supported and is expected to continue to work in most cases. However, some limitations may exist. One known limitation is that LiveCompare requires the `comparison_algorithm` parameter to be set to `full_row` (for example, `comparison_algorithm = full_row`).

PgBouncer support

You can use LiveCompare against nodes through PgBouncer. However, you must use `pool_mode=session` because LiveCompare uses prepared statements on PostgreSQL, which isn't possible when `pool_mode` is either `transaction` or `statement`.

5 Command-line usage

Compare mode

Copy any `/etc/livecompare/template*.ini` to use in your project and adjust as necessary. See Settings.

```
cp /etc/livecompare/template_basic.ini my_project.ini

livecompare my_project.ini
```

While LiveCompare executes, N+1 progress bars appear, where N is the number of processes. (You can specify the number of processes in the settings.) The first progress bar shows overall execution. The other progress bars show the current table being processed by a specific process.

The information being shown for each table is, from left to right:

- Number of the process
- Table name
- Status, which can be the ID of the comparison round followed by the current table chunk.

  `p1/1` means the table wasn't split. A status of `setup` means the table is being analyzed (checking row count and splitting if necessary).
- Number of rows processed
- Number of total rows being considered in this comparison round
- Time elapsed
- Estimated time to complete
- Speed in records per second

When table splitting is enabled (`parallel_chunk_rows > 0`), if a table has more rows than the `parallel_chunk_rows` setting, then a hash function is used to determine the job that considers each row. This can slow down the comparison individually. However the comparison as a whole might benefit from parallelism for the given table.

While the program is executing, you can cancel it at any time by pressing **Ctrl-C**. A message like the following appears:

```
 Manually stopping session 6... You can resume the session with:

livecompare my_project.ini 6
```

> **Important**
>
> If LiveCompare is running in the background or running in another shell, you can still softly stop it. It keeps the `PID` of the master process inside the session folder (`lc_session_6` in the example) in a file named `livemaster.pid`. You can then invoke `kill -2 <PID>` to softly stop it.

Then, at any time you can resume a previously canceled session, for example:

```
livecompare my_project.ini 6
```

When the program ends, if it found no inconsistencies, the output is similar to the following:

```
 Saved file lc_session_5/summary_20190514.out with the complete table summary.
You can also get the table summary by connecting to the output database and executing:
select * from livecompare.vw_table_summary where session_id = 5;

Elapsed time: 0:02:10.970954
Processed 3919015 rows in 6 tables using 3 processes.
Found 0 inconsistent rows in 0 tables.
```

If any inconsistencies were found, the output looks like this:

```
 Comparison finished, waiting for remaining difference checks...

Outstanding differences:

+--------------+-------------------+-----------------+------------------+----------------------+------------
----+----------------------------+
|   session_id | table_name        | elapsed_time    |   num_total_rows |   num_processed_rows |
num_differences |   max_num_ignored_columns |
|--------------+-------------------+-----------------+------------------+----------------------+------------
----+----------------------------|
|            6 | public.categories | 00:00:00.027864 |               18 |                   18 |
4 |                            |
+--------------+-------------------+-----------------+------------------+----------------------+------------
----+----------------------------+

Saved file lc_session_6/summary_20200129.out with the complete table summary.
You can also get the table summary by connecting to the output database and executing:
select * from livecompare.vw_table_summary where session_id = 6;

Elapsed time: 0:00:50.149987
Processed 172718 rows in 8 tables from 3 connections using 2 workers.
Found 4 inconsistent rows in 1 tables.

Saved file lc_session_6/differences_20200129.out with the list of differences per table.
You can also get a list of differences per table with:
select * from livecompare.vw_differences where session_id = 6;
Too see more details on how LiveCompare determined the differences:
select * from livecompare.vw_consensus where session_id = 6;

Script lc_session_6/apply_on_the_first_20200129.sql was generated, which can be applied to the first
connection and make it consistent with the majority of connections.
```

```
You can also get this script with:
select difference_fix_dml from livecompare.vw_difference_fix where session_id = 6 and connection_id =
'first';
```

Recheck mode

In a PGD environment, any divergence that PGD finds can later not exist, as the replication caught up due to eventual consistency. Depending on several factors, replication lag can cause LiveCompare to report false positives.

To overcome that, in a later moment when replication lag has decreased or data has already caught up, you can manually execute a recheck only on the differences that were previously found. This execution mode is called *recheck*. You can execute it like this:

```
livecompare my_project.ini 6 --recheck
```

In this mode, LiveCompare generates separate recheck logs and updates all reports that already exist in the `lc_session_X` directory.

> **Important**
>
> If resuming a `compare` or executing under `recheck`, LiveCompare checks whether the settings and connections attributes are the same as when the session was created. If any divergence is found, it quits the execution and gives a message.

Conflicts mode

To run LiveCompare in `conflicts` mode, invoke it with:

```
livecompare my_project.ini --conflicts
```

For more details about the `conflicts` mode, see PGD support.

Dry-run mode

> **New Feature**
>
> LiveCompare dry-run mode support is available for LiveCompare version 2.2.0 and later.

For example, suppose you have the following INI file:

```
[General Settings]
logical_replication_mode = off
difference_tie_breakers = first

[First
Connection]
dsn = dbname=testb

[Second
Connection]
```

```
dsn =
dbname=testdb2

[Third
Connection]
dsn =
dbname=testdb3

[Output
Connection]
dsn =
dbname=liveoutpu

[Table Filter]
schemas = schema_name =
'public'
```

As the DSN under `Output Connection` (the LiveCompare cache database) is incorrect, running LiveCompare initially fails with:

```
Output connection is not reachable.
```

After fixing this, then the output connection is now reachable. But suppose that only one of the data connections is set correctly. In that case, LiveCompare fails again with:

```
At least two reachable connections are required.
Following connections are unreachable: first, third.
Following connections are reachable: second.
```

LiveCompare can start a comparison with at least two data connections available. So you go ahead and fix the third connection. But LiveCompare still fails with:

```
A difference_tie_breakers host is not a reachable connection: first.
```

This happens because the example set `difference_tie_breakers = first`, and any connection set as a tie breaker or source of truth needs to be reachable.

After fixing all those issues, then LiveCompare can start the comparison.

However, when setting up a comparison from scratch, you can check beforehand whether LiveCompare will abort with a configuration error. Further checks of this nature are all shown in the order LiveCompare performs them.

You can do this with the `--dry-run` mode, which:

- Prints all execution aborts that will happen due to configuration issues.
- Prints the list of connections with some details, including if it's reachable.
- Prints the table filter.
- After applying the table filter, prints the list of tables that are common to the reachable connections.

Here's one sample output, given the example `.ini` file, and all configuration errors regarding unreachable connections:

```
$ livecompare test.ini --dry-run
EnterpriseDB LiveCompare 2.2.0, dry-run mode


Output connection is not reachable.

At least two reachable connections are required.
Following connections are unreachable: first, third.
```

```
Following connections are reachable: second.

A difference_tie_breakers host is not a reachable connection: first.


Connections

+--------+--------------+-----------+---------------+---------------------+-----------+---------------+-----
------------+-------------+
| ID     | Technology   | Version   | PGD Version   | Pglogical Version   | Initial   | Tie Breaker   |
Source of Truth   | Reachable   |
|--------+--------------+-----------+---------------+---------------------+-----------+---------------+-----
------------+-------------|
| second | postgresql   | 110015    | -             | -                   | False     | False         |
False             | True        |
| first  | postgresql   | -         | -             | -                   | False     | True          | Fals
| False       |
| third  | postgresql   | -         | -             | -                   | False     | False         | Fals
| False       |
| output | postgresql   | -         | -             | -                   | -         | -             | -
| False       |
+--------+--------------+-----------+---------------+---------------------+-----------+---------------+-----
------------+-------------+


Table Filter

publications = ''
replication_sets = ''
schemas = schema_name = 'public'
tables = ''


Tables

+---------------------+--------------+-----------------+--------------------------+
| Table Name          | Row Filter   | Column Filter   | Custom Comparison Key    |
|---------------------+--------------+-----------------+--------------------------|
| public.categories   | -            | -               | -                        |
| public.cust_hist    | -            | -               | -                        |
| public.customers    | -            | -               | -                        |
| public.departments  | -            | -               | -                        |
| public.dept_emp     | -            | -               | -                        |
| public.dept_manager | -            | -               | -                        |
| public.employees    | -            | -               | -                        |
| public.inventory    | -            | -               | -                        |
| public.orderlines   | -            | -               | -                        |
| public.orders       | -            | -               | -                        |
| public.products     | -            | -               | -                        |
| public.reorder      | -            | -               | -                        |
| public.salaries     | -            | -               | -                        |
| public.tbl          | -            | -               | -                        |
| public.titles       | -            | -               | -                        |
+---------------------+--------------+-----------------+--------------------------+
```

6 Advanced usage

When LiveCompare runs, it creates a folder called `lc_session_<session_id>` in the working directory. This folder contains the following files:

- `lc_<execution_mode>_<current_date>.log` — Log file for the session.
- `summary_<current_date>.out` — A list of all tables that were processed. For each table, it shows the time LiveCompare took to process the table, the total number of rows and how many rows were processed, how many differences were found in the table, and the maximum number of ignored columns, if any.

  To get the complete summary, you can also execute the following query against the output database:

  ```
  select *
  from <output_schema>.vw_table_summary
  where session_id = <session_id>;
  ```

- `differences_<current_date>.out` — Useful information about any differences. This file isn't generated if there are no differences.

  The following is an example of a difference list:

```
+-------------------+--------------------------+-----------------+----------------------+
| table_name        | table_pk_column_names    |   difference_pk | difference_status    |
|-------------------+--------------------------+-----------------+----------------------|
| public.categories | category                 |             (7) | P                    |
| public.categories | category                 |            (10) | P                    |
| public.categories | category                 |            (17) | P                    |
| public.categories | category                 |            (18) | P                    |
+-------------------+--------------------------+-----------------+----------------------+
```

To get the full list of differences with all details, you can execute the following query against the output database:

```
```postgresql
select *
from <output_schema.vw_differences
where session_id = <session_id>;
```

To understand how LiveCompare consensus worked to decide which databases are divergent, the view
`vw_consensus` can provide details on the consensus algorithm:

```postgresql
select *
from <output_schema.vw_consensus
where session_id = <session_id>;
```
```

- `apply_on_the_first_<current_date>.sql` — If there are any differences, this file shows a DML command to apply on the first database to make it consistent with all other databases. The following is an example of a script for the differences shown in the table:

  ```
  BEGIN;

  DELETE FROM public.categories WHERE (category) = 7;
  UPDATE public.categories SET categoryname = $lc1$Games Changed$lc1$ WHERE (category) = 10;
  INSERT INTO public.categories (category,categoryname) VALUES (17, $lc1$Test 1$lc1$);
  INSERT INTO public.categories (category,categoryname) VALUES (18, $lc1$Test 2$lc1$);

  COMMIT;
  ```

LiveCompare generates this script. To fix the inconsistencies in the first database, execute the script in it.

LiveCompare generates a similar `apply_on_*.sql` script for each database that has inconsistent data.

Aborting comparisons

Before starting the comparison session, LiveCompare tries all connections. If the number of reachable connections isn't at least two, then LiveCompare aborts the whole session and gives an error message. If at least two connections are reachable, then LiveCompare proceeds with the comparison session. For all connections, LiveCompare writes a flag `connection_reachable` in the `connections` table in the cache database.

For all reachable connections, LiveCompare does some sanity checks around the database technologies and the setting `logical_replication_mode`. If any of the sanity checks fail, then LiveCompare aborts the comparison and gives an error message.

Considering the tables available on all reachable connections, LiveCompare builds the list of tables to compare, taking into account the table filter. If a specific table doesn't exist on at least two connections, then the comparison on that specific table is aborted.

LiveCompare initially gathers metadata from all tables. This step is called *setup*. If any errors happen during the setup, for example, the user doesn't have access to a specific table, then it's called a *setup error*. If `abort_on_setup_error` is enabled, then LiveCompare aborts the whole comparison session, and the program finishes with an error message. Otherwise, only the table having the error has its table comparison aborted, and LiveCompare moves on to the next table.

For each table that LiveCompare starts the table comparison on, LiveCompare first checks the table definition on all reachable connections. If the tables don't have the same columns and column data types, LiveCompare applies `column_intersection`. If there are no columns to compare, then LiveCompare aborts the table comparison.

Comparison key

For each table being compared, when gathering the table metadata, LiveCompare builds the comparison key to use in the table comparison, following these rules:

1. Use the custom comparison key if configured.
2. Alternatively, use PK if available.
3. Alternatively, if the table has `UNIQUE` indexes, among the `UNIQUE` indexes that have all `NOT NULL` columns, use the `UNIQUE` index with fewer columns.
4. If none of these are possible, try to use all `NOT NULL` columns as a comparison key. `NULL` columns are also considered if `ignore_nullable = false`.

If you decide to use strategies 1 or 4 as a comparison key, then LiveCompare also checks for uniqueness on the key. If uniqueness isn't possible, then LiveCompare aborts the comparison on that table. You can disable this behavior by using `check_uniqueness_enforcement = false`.

Differences to fix

LiveCompare can identify and provide fixes for the following differences:

- A row exists in the majority of the data connections. The fix is an `INSERT` on the divergent databases.
- A row doesn't exist in the majority of the data connections. The fix is a `DELETE` on the divergent databases.

- A row exists in all databases, but some column values mismatch. The fix is an `UPDATE` on the divergent databases.

The default setting is `difference_statements = all`, which means that LiveCompare tries to apply all three DML types (`INSERT`, `UPDATE`, and `DELETE`) for each difference it finds. But you can specify the type of DML for LiveCompare to consider when providing difference fixes. Change the value of the setting `difference_statements` to any of these values:

- `all` (default): Fixes `INSERT`, `UPDATE`, and `DELETE` DML types.
- `inserts`: Fixes only `INSERT` DML types.
- `updates`: Fixes only `UPDATE` DML types.
- `deletes`: Fixes only `DELETE` DML types.
- `inserts_updates`: Fixes only `INSERT` and `UPDATE` DML types.
- `inserts_deletes`: Fixes only `INSERT` and `DELETE` DML types.
- `updates_deletes`: Fixes only `UPDATE` and `DELETE` DML types.

When `difference_statements` has the values `all`, `updates`, `inserts_updates`, or `updates_deletes`, then you can tell LiveCompare to ignore any `UPDATE` that sets `NULL` to a column.

Difference log

The table `difference_log` stores all information about differences every time LiveCompare checks them. You can run LiveCompare in recheck mode multiple times, so this table shows how the difference evolved over the time window in which LiveCompare was rechecking it.

- **Detected (D)**: The difference was just detected. In recheck and fix modes, LiveCompare marks all Permanent and Tie differences as Detected so it can recheck them.
- **Permanent (P)**: After rechecking the difference, if data is still divergent, LiveCompare marks the difference as Permanent.
- **Tie (T)**: This entry is the same as Permanent, but there isn't enough consensus to determine the connections that are the majority.
- **Absent (A)**: If, upon a recheck, LiveCompare finds that the difference doesn't exist anymore, that is, the row is now consistent between both databases, then LiveCompare marks the difference as Absent.
- **Volatile (V)**: If, upon a recheck, `xmin` changed on an inconsistent row, then LiveCompare marks the difference as Volatile.
- **Ignored (I)**: You can stop difference recheck of certain differences by manually calling the function `<livecompare_schema_name>.accept_divergence(session_id, table_name, difference_pk)` in the output PostgreSQL connection. For example:

```
SELECT livecompare.accept_divergence(
    2                  -- session_id
  , 'public.categories' -- table_name
  , $$(10)$$           -- difference_pk
);
```

7 EDB Postgres Distributed support

You can use LiveCompare against EDB Postgres Distributed (PGD, formerly known as BDR) nodes as well as non-PGD nodes.

Setting `logical_replication_mode = bdr` makes the tool assume that all databases being compared belong to the same PGD cluster. Then you can specify node names as connections and replication sets to filter tables.

For example, suppose you can connect to any node in the PGD cluster, which we'll refer to as the initial connection. By initially connecting to this node, LiveCompare can check PGD metadata and retrieve connection information from all other nodes.

Now suppose you want to compare three PGD nodes. As LiveCompare can connect to any node starting from the initial connection, you don't need to define `dsn` or any connection information for the data connections. You only need to define `node_name`. LiveCompare searches in PGD metadata about the connection information for that node and then connects to the node.

For LiveCompare to connect to all other nodes by fetching PGD metadata, LiveCompare must be able to connect to them using the same DSN from the PGD view `bdr.node_summary` in the field `interface_connstr`. In this case, we recommend running LiveCompare on the same machine as the initial connection as the postgres user. If that's not possible, then define the `dsn` attribute in all data connections.

You can also specify replication sets as table filters. LiveCompare uses PGD metadata to build the table list, considering only tables that belong to the replication sets you defined in the `replication_sets` setting.

For example, you can create an `.ini` file to compare three PGD nodes:

```
[General Settings]
logical_replication_mode =
bdr
max_parallel_workers = 4

[Initial
Connection]
dsn = port=5432 dbname=live
user=postgres

[Node1
Connection]
node_name = node1

[Node2
Connection]
node_name = node2

[Node3
Connection]
node_name = node3

[Output
Connection]
dsn = port=5432 dbname=liveoutput user=postgres

[Table Filter]
replication_sets = set_name =
'bdrgroup'
```

You can also tell LiveCompare to compare all active nodes in the PGD cluster. To do so:

1. Under `General Settings`, enable `all_bdr_nodes = on`.
2. Under `Initial Connection`, specify an initial connection.

Additional data connections aren't required.

For example:

```
[General Settings]
logical_replication_mode =
bdr
max_parallel_workers = 4
all_bdr_nodes = on
```

```
[Initial
Connection]
dsn = port=5432 dbname=live
user=postgres

[Output
Connection]
dsn = port=5432 dbname=liveoutput user=postgres

[Table Filter]
replication_sets = set_name =
'bdrgroup'
```

When `all_bdr_nodes = on`, LiveCompare uses the `Initial Connection` setting to fetch the list of all PGD nodes. While additional data connections aren't required, if set, they're appended to the list of data connections. For example, you can compare a whole PGD cluster against a single Postgres connection, which is useful in migration projects:

```
 [General Settings]
logical_replication_mode =
bdr
max_parallel_workers = 4
all_bdr_nodes = on

[Initial
Connection]
dsn = port=5432 dbname=live
user=postgres

[Old
Connection]
dsn = host=oldpg port=5432 dbname=live
user=postgres

[Output
Connection]
dsn = port=5432 dbname=liveoutput user=postgres

[Table Filter]
replication_sets = set_name =
'bdrgroup'
```

The settings `node_name` and `replication_sets` are supported for the following technologies:

- PGD 1, 2, 3, and 4
- pglogical 2 and 3

To enable pglogical metadata fetch instead of PGD, set `logical_replication_mode = pglogical` instead of `logical_replication_mode = bdr`.

PGD witness nodes

Using replication sets in PGD, you can configure specific tables to include in the PGD replication. You can also specify the nodes to receive data from these tables by configuring the node to subscribe to the replication set the table belongs to. This setting allows for different architectures such as PGD sharding and the use of PGD witness nodes.

A PGD witness is a regular PGD node that doesn't replicate any DML from other nodes. The purpose of the witness is to provide quorum in Raft Consensus voting. (For details on the PGD witness node, see Witness nodes in the PGD documentation.) Replication set configuration determines whether the witness replicates DDLs. This means that there are two types of PGD witnesses:

- A completely empty node, without any data nor tables

- A node that replicates DDL from other nodes, so it has empty tables

In the first case, even if the PGD witness is included in the comparison (either manually under `[Connections]` or using `all_bdr_nodes = on`), because the witness doesn't have any tables, the following message is logged:

```
Table public.tbl does not exist on connection node1
```

In the second case, the table exists on the PGD witness. However, it's not correct to report data missing on the witness as divergences. So, for each table, LiveCompare checks the following information on each node included in the comparison:

- The replication sets that the node subscribes to
- The replication sets that the table is associated with
- The replication sets, if any, you defined in the filter `replication_sets` under `Table Filter`

If the intersection among all three lists of replication sets is empty, which is the case for the PGD witness, then LiveCompare logs this message:

```
Table public.tbl is not subscribed on connection node1
```

In both cases, the comparison for that table proceeds on the nodes where the table exists, and the table is replicated according to the replication sets configuration.

Differences in a PGD cluster

LiveCompare makes changes only to the local node. It's important that corrective changes don't get replicated to other nodes.

When `logical_replication_mode = bdr`, LiveCompare first checks if a replication origin called `bdr_local_only_origin` already exists. (You can configure the name of the replication origin by adjusting the setting `difference_fix_replication_origin`.) If a replication origin called `bdr_local_only_origin` doesn't exist, then LiveCompare creates it on all PGD connections.

> **Important**
>
> PGD 3.6.18 introduced the new preexisting `bdr_local_only_origin` replication origin to use for applying local-only transactions. If LiveCompare is connected to PGD 3.6.18, it doesn't create this replication origin.

LiveCompare generates apply scripts considering the following:

- Set the current transaction to use the replication origin `bdr_local_only_origin`, so any DML executed has `xmin` associated with `bdr_local_only_origin`.
- Set the current transaction datetime to be far in the past, so if there are any PGD conflicts with real DML being executed on the database, LiveCompare DML always loses the conflict.

After applying a LiveCompare fix script to a PGD node, you can get exactly the rows that were inserted or updated by LiveCompare using the following query. Replace `mytable` with the name of any table.

```
with lc_origin as (
    select roident
    from pg_replication_origin
    where roname = 'bdr_local_only_origin'
)
select t.*
from mytable t
inner join lc_origin r
on r.roident = bdr.pg_xact_origin(t.xmin);
```

Deleted rows are no longer visible.

LiveCompare requires at least a PostgreSQL user with bdr_superuser privileges to properly fetch metadata.

All of these steps involving replication origins applied only to the output script if the PostgreSQL user has bdr_superuser or PostgreSQL superuser privileges. Otherwise, LiveCompare generates fixes without associating any replication origin. (Transaction replication is still disabled using `SET LOCAL bdr.xact_replication = off`.) However, we recommend using a replication origin when applying the DML scripts. Otherwise, LiveCompare has the same precedence as a regular user application regarding conflict resolution. Also, as there isn't any replication origin associated with the fix, you can't use the query to list all rows fixed by LiveCompare.

Between PGD 3.6.18 and PGD 3.7.0, the following functions are used:

- `bdr.difference_fix_origin_create()`: Executed by LiveCompare to create the replication origin specified in `difference_fix_replication_origin` (by default, set to `bdr_local_only_origin`), if this replication origin doesn't exist.
- `bdr.difference_fix_session_setup()`: Included in the generated DML script so the transaction is associated with the replication origin specified in `difference_fix_replication_origin`.
- `bdr.difference_fix_xact_set_avoid_conflict()`: Included in the generated DML script so the transaction is set far in the past (`2010-01-01`). The fix transaction applied by LiveCompare always loses any conflict.

These functions require a bdr_superuser rather than a PostgreSQL superuser. Starting with PGD 3.7.0, those functions are deprecated. In that case, if running as a PostgreSQL superuser, LiveCompare uses the following functions to perform the same actions:

- `pg_replication_origin_create(origin_name)`;
- `pg_replication_origin_session_setup()`;
- `pg_replication_origin_xact_setup()`.

If a PostgreSQL superuser isn't being used, then LiveCompare includes only the following in the generated DML transaction:

```
SET LOCAL bdr.xact_replication = off;
```

Conflicts in PGD

LiveCompare has an execution mode called `conflicts`. This execution mode is specific for PGD clusters. It works only in PGD 3.6, PGD 3.7, PGD 4, and PGD 5 clusters.

While `compare` mode is used to compare all content of tables as a whole, `conflicts` mode focuses just in tuples/tables that are related to existing conflicts that are registered in `bdr.apply_log`, in case of PGD 3.6, or in `bdr.conflict_history`, in case of PGD 3.7, PGD 4, and PGD 5.

`conflicts` execution mode is expected to run much faster than `compare` mode because it inspects only specific tuples from specific tables. However, it's not as complete as `compare` mode for the same reason.

The main objective of this execution mode is to check that the automatic conflict resolution that's being done by PGD is consistent among nodes, that is, after PGD resolves conflicts, the cluster is in a consistent state.

Although, for the general use case, automatic conflict resolution ensures cluster consistency, there are a few known cases where automatic conflict resolution can result in divergent tuples among nodes. So the `conflicts` execution mode from LiveCompare can help with checking and ensuring consistency, providing a good balance between time and result.

Conflict example

Suppose on `node3`, you execute the following query:

```
SELECT c.reloid::regclass,
       s.origin_name,
       c.local_time,
       c.key_tuple,
       c.local_tuple,
       c.remote_tuple,
       c.apply_tuple,
       c.conflict_type,
       c.conflict_resolution
FROM bdr.conflict_history c
INNER JOIN bdr.subscription_summary s
ON s.sub_id = c.sub_id;
```

You can see the following conflict in `bdr.conflict_history`:

```
reloid              | tbl
origin_name         | node2
local_time          | 2021-05-13 19:17:43.239744+00
key_tuple           | {"a":null,"b":3,"c":null}
local_tuple         |
remote_tuple        |
apply_tuple         |
conflict_type       | delete_missing
conflict_resolution | skip
```

This conflict means that when the `DELETE` arrived from `node2` to `node3`, there was no row with `b = 3` in table `tbl`. However, the `INSERT` might have arrived from `node1` to `node3` later, which then added the row with `b = 3` to `node3`. So this is the current situation on `node3`:

```
bdrdb=# SELECT * FROM tbl WHERE b = 3;
 a | b |  c
---+---+-----
 x | 3 | foo
(1 row)
```

While on nodes `node1` and `node2`, you see this:

```
bdrdb=# SELECT * FROM tbl WHERE b = 3;
 a | b | c
---+---+---
(0 rows)
```

The PGD cluster is divergent.

To detect and fix such divergence, you can execute LiveCompare in `compare` mode. However, depending on the size of the comparison set (suppose table `tbl` is very large), that can take a long time, even hours.

This situation is one in in which `conflicts` mode can be helpful. In this case, the `delete_missing` conflict is visible only from `node3`, but LiveCompare can extract the PK values from the conflict logged rows (`key_tuple`, `local_tuple`, `remote_tuple`, and `apply_tuple`) and perform an automatic cluster-wide comparison only on the affected table, already filtering by the PK values. The comparison then checks the current row version in all nodes in the cluster.

Create a `check.ini` file to set `all_bdr_nodes = on`, that is, to tell LiveCompare to compare all nodes in the cluster:

```
[General Settings]
logical_replication_mode = bdr
max_parallel_workers = 2
```

```
all_bdr_nodes = on

[Initial Connection]
dsn = dbname=bdrdb

[Output Connection]
dsn = dbname=liveoutput
```

To run LiveCompare in `conflicts` mode:

```
livecompare check.ini --conflicts
```

After the execution, in the console output, you see something like this:

```
 Elapsed time: 0:00:02.443557
Processed 1 conflicts about 1 tables from 3 connections using 2 workers.
Found 1 divergent conflicts in 1 tables.
Processed 1 rows in 1 tables from 3 connections using 2 workers.
Found 1 inconsistent rows in 1 tables.
```

Inside folder `./lc_session_X/` (`X` is the number of the current comparison session), LiveCompare writes the file `conflicts_DAY.out` (replacing `DAY` in the name of the file with the current day). The file shows the main information about all divergent conflicts.

If you connect to database `liveoutput`, you can see more details about the conflicts, for example, using this query:

```
 SELECT *
FROM livecompare.vw_conflicts
WHERE session_id = 1
  AND conflict_id = 1
ORDER BY table_name,
         local_time,
         target_node;
```

The output is something like this:

```
 session_id             | 1
table_name              | public.tbl
conflict_id             | 1
connection_id           | node3
origin_node             | node2
target_node             | node3
local_time              | 2021-05-13 19:17:43.239744+00
key_tuple               | {"a": null, "b": 3, "c": null}
local_tuple             |
remote_tuple            |
apply_tuple             |
conflict_type           | delete_missing
conflict_resolution     | skip
conflict_pk_value_list | {(3)}
difference_log_id_list | {1}
is_conflict_divergent   | t
```

The `is_conflict_divergent = true` means that LiveCompare compared the conflict and found the nodes to be currently divergent in the tables and rows reported by the conflict. The view `livecompare.vw_conflicts` shows information about all conflicts, including the non-divergent ones.

LiveCompare also generates the DML script `./lc_session_X/apply_on_the_node3_DAY.sql` (where `DAY` in the name of the file with the

current day):

```
BEGIN;

SET LOCAL bdr.xact_replication = off;
SELECT pg_replication_origin_session_setup('bdr_local_only_origin');
SELECT pg_replication_origin_xact_setup('0/0', '2010-01-01'::timestamptz);;

SET LOCAL ROLE postgres;
DELETE FROM public.tbl WHERE (b) = (3);

COMMIT;
```

LiveCompare is suggesting to `DELETE` the row where `b = 3` from `node3` because the row doesn't exist on the other two rows. By default, LiveCompare suggests the DML to fix based on the majority of the nodes.

Running this DML script against `node3` makes the PGD cluster consistent again:

```
psql -h node3 -f ./lc_session_X/apply_on_the_node3_DAY.sql
```

As the `--conflicts` mode comparison is much faster than a full `--compare`, we strongly recommend scheduling a `--conflicts` comparison session more often to ensure conflict resolution is providing cluster-wide consistency.

> **Note**
>
> To see the data in `bdr.conflict_history` in PGD 3.7 or `bdr.apply_log` in PGD 3.6, run LiveCompare with a user that's a bdr_superuser or a PostgreSQL superuser.

To be able to see the data in `bdr.conflict_history` in PGD 3.7+ or `bdr.apply_log` in PGD 3.6, run LiveCompare with a user that's bdr_superuser or a PostgreSQL superuser.

Conflicts Filter

You can also tell LiveCompare to filter the conflicts by any of the columns in either `bdr.conflicts_history` or `bdr.apply_log`. For example:

```
[Conflicts Filter]
conflicts = table_name = 'public.tbl' and conflict_type =
'delete_missing'
```

Mixing technologies

Metadata for `node_name` and `replication_sets` are fetched in the initial connection. So it must be a pglogical- and/or PGD-enabled database.

The list of tables is built in the first data connection. So the `replication_sets` condition must be valid in the first connection.

You can perform mixed-technology comparisons, for example:

- PGD 1 node versus PGD 3 node
- PGD 4 node versus vanilla Postgres instance
- Vanilla Postgres instance versus pglogical node

8 Oracle support

You can use LiveCompare to compare data from an Oracle database against any number of PostgreSQL or PGD databases.

For example, you can define `technology = oracle` in a data connection. You can then use other settings to define the connection to Oracle:

- `host`
- `port`
- `service`
- `user`
- `password`

All other data connections must be PostgreSQL.

Here's a simple example of comparison between an Oracle database and a PostgreSQL database:

```
[General Settings]
logical_replication_mode = off
max_parallel_workers = 4
oracle_user_tables_only = on
oracle_ignore_unsortable = on
column_intersection = on
force_collate =
C
difference_tie_breakers =
oracle

[Oracle
Connection]
technology =
oracle
host = 127.0.0.1
port = 1521
service = XE
user = LIVE
password = live

[Postgres
Connection]
technology = postgresql
dsn = dbname=liveoracle user=william

[Output
Connection]
dsn = dbname=liveoutput user=william

[Table Filter]
schemas = schema_name =
'live'
```

Here, the `schema_name` in Oracle is the user table sandbox. All table names are schema qualified by default:

- Postgres: `<schema_name> . <table_name>`
- Oracle: `<user> . <table_name`

You can disable schema-qualified table names by setting `schema_qualified_table_names = off`. You can do this only if `oracle_user_tables_only = on`. This setting tells LiveCompare to search only on tables that belong to the Oracle user that's connected. When schema-qualified table names is disabled, then on Postgres you need to have set a default `search_path` on your role or configuration. Or, you can use the connection `start_query` parameter to set an appropriate `search_path`, for example:

```
[General Settings]
logical_replication_mode = off
max_parallel_workers = 4
oracle_user_tables_only = on
oracle_ignore_unsortable = on
column_intersection = on
force_collate =
C
difference_tie_breakers =
oracle
schema_qualified_table_names = off

[Oracle
Connection]
technology =
oracle
host = 127.0.0.1
port = 1521
service = XE
user = LIVE
password = live

[Postgres
Connection]
technology = postgresql
dsn = dbname=liveoracle user=william
start_query = SET search_path = myschema1, myschema2,
public

[Output
Connection]
dsn = dbname=liveoutput user=william

[Table Filter]
tables = table_name in ('mytable1',
'mytable2')
```

When `schema_qualified_table_names = off`, you can also use non-qualified table names in `Table Filter`, `Row Filter`, and `Column Filter`.

> **Note**
>
> The `Output Connection` is required to write progress and reporting information from LiveCompare.

If you need to compare a PGD database against Oracle, and you want to take advantage of `Initial Connection`, `node_name`, and `replication_sets` features (described in PGD support), then you can point the last data connection to Oracle, like this:

```
[General Settings]
logical_replication_mode =
bdr
max_parallel_workers = 4
oracle_user_tables_only = on
oracle_ignore_unsortable = on
column_intersection = on
force_collate =
C
difference_tie_breakers =
oracle

[Initial
Connection]
dsn = port=5432 dbname=live
user=postgres
```

```
[BDR
Connection]
node_name = node1

[Oracle
Connection]
technology =
oracle
host = 127.0.0.1
port = 1521
service = XE
user = LIVE
password = live

[Output
Connection]
dsn = port=5432 dbname=liveoutput user=postgres

[Table Filter]
replication_sets = set_name =
'bdrgroup'
```

You also can compare a whole PGD cluster against a single Oracle database, for example:

```
 [General Settings]
logical_replication_mode =
bdr
max_parallel_workers = 4
oracle_user_tables_only = on
oracle_ignore_unsortable = on
column_intersection = on
force_collate =
C
difference_tie_breakers =
oracle
all_bdr_nodes = on

[Initial
Connection]
dsn = port=5432 dbname=live
user=postgres

[Oracle
Connection]
technology =
oracle
host = 127.0.0.1
port = 1521
service = XE
user = LIVE
password = live

[Output
Connection]
dsn = port=5432 dbname=liveoutput user=postgres

[Table Filter]
replication_sets = set_name =
'bdrgroup'
```

Requirements

LiveCompare works on PostgreSQL databases out-of-the-box. You don't need to install any additional software. But to be able to connect to Oracle, LiveCompare does requires additional software.

Oracle Instant Client

You need to download and install Oracle Instant Client (or extract it to a specific folder, depending on the operating system you use):

- **MacOSX**: Download Oracle Instant Client (64-bit) and extract in `~/lib`;
- **Linux**: Download Oracle Instant Client (32-bit) (64-bit) and install it on your system, then set `LD_LIBRARY_PATH`;
- **Windows**: Download Oracle Instant Client (32-bit) (64-bit) and extract it into the LiveCompare folder.

cx_Oracle Python module

You need the Python module cx_Oracle installed and available on your system so that LiveCompare can connect to an Oracle database.

Currently, cx_Oracle isn't installable from Linux distribution repositories, so follow the instructions on the cx_Oracle website to install it on your system.

We recommend executing LiveCompare under the postgres operating system user. Then you can install the `cx_Oracle` module through PIP only for the `postgres` user, using the following command:

```
pip3 install --user cx_Oracle --upgrade
```

Differences

If LiveCompare finds any difference, it generates a DML script to apply only on the PostgreSQL connections. No DML script to apply on the Oracle connection is generated.

BLOB and CLOB data types

LiveCompare can compare CLOB fields from Oracle, provided that the equivalent field in PostgreSQL is of type `text`. The same goes for BLOB fields from Oracle. The equivalent in PostgreSQL is of type `bytea`.

However, by default, LiveCompare doesn't handle BLOB/CLOB fields if they're in the primary key or if the table has no primary key. If that's the case, then the table is ignored, and LiveCompare logs has a message like this:

```
ORA-00932: inconsistent datatypes: expected - got BLOB
```

You can work around this behavior by telling LiveCompare to ignore BLOB/CLOB fields if the table has no primary key. Enable these two settings in the `General Settings` section:

```
oracle_ignore_unsortable = on
column_intersection = on
```

Incompatible collation

On Oracle, generally the following initialization parameters are set:

```
NLS_COMP = BINARY
NLS_SORT = BINARY
```

This means that, regardless of the `NLS_LANG` and other language settings, all `ORDER BY` operations in Oracle are performed using the character binary code.

In Postgres, the equivalent collation that shows the same behavior is the `C` collation. If your Postgres database was initialized in a different collation, then by default LiveCompare might find issues when sorting PK values. This can lead to false positives.

To work around that, you can force a collation (say, the `C` collation) in Postgres so it matches the same sort behavior from Oracle:

```
force_collate =
C
```

If LiveCompare detects that the comparison session involves Oracle and PostgreSQL, then LiveCompare already sets `force_collate = C`, unless you set it to another value.

Common hash

By default, LiveCompare has `comparison_algorithm = block_hash`, even when comparing PostgreSQL to Oracle. However, a *common hash* is built following these rules:

- The row is fully represented as text by concatenating all column values.
- On the Postgres side, timestamp, numeric, and bytea data types are handled to mimic Oracle.
- This way, the full row representation is then hashed using MD5 on both sides.
- This allows using `comparison_algorithm` set to `block_hash` and `row_hash`.
- If there are any mismatches when using `block_hash`, LiveCompare falls back to `row_hash` and then `full_row`, as it does in a Postgres versus Postgres comparison.
- The BLOB, CLOB, and NCLOB fields on Oracle are limited to only the first 2000 characters. `comparison_algorithm = full_row` allows comparison of the entire BLOB and CLOB.
- On Oracle, the full row representation must not be wider than 4000 characters. If the full row representation is wider than 4000 characters, LiveCompare aborts the comparison for that specific table, and the following error message is added to the logs:

```
ORA-01489: result of string concatenation is too long
```

Later LiveCompare versions will fall back to `full_row` comparison on these specific tables. For now, a workaround is to configure a separate comparison sessions only on these tables, using `comparison_algorithm = full_row`. Using LiveCompare with Oracle 10g always requires `comparison_algorithm = full_row` be set.

The common hash uses the `standard_hash` function on Oracle 12c and later. On Oracle 11g, the `standard_hash` function isn't available, so LiveCompare tries to use the `dbms_crypto.hash` function instead. However, it might require additional privileges for the user on the Oracle side, for example:

```
GRANT EXECUTE ON sys.dbms_crypto TO testuser;
```

9 Settings

General settings

- `logical_replication_mode`: Affects how the program interprets connections and table filter settings and also the requirements to check for in the connections before starting the comparison. Currently the possible values are:
  - `off`: Assumes there's no logical replication between the databases.
  - `native`: Assumes there's native logical replication between the databases. Enables the use of the `Table Filter -> publications` setting to specify the list of tables to use. Requires PostgreSQL 10+ on all databases.
  - `pglogical`: Assumes there's pglogical replication between the databases. Enables the use of the `Table Filter -> replication_sets` setting to specify the list of tables to use. Also enables the use of `node_name` to specify the data connections, which requires setting the `Initial Connection` that's used to retrieve DSN information of the nodes. Requires the `pglogical` extensions to be installed on all databases.
  - `bdr`: Assumes all data connections are nodes from the same PGD cluster. Enables use of the `Table Filter -> replication_sets` setting to specify the list of tables to use. Also enables the use of `node_name` to specify the data connections, which requires setting the `Initial Connection` that's used to retrieve DSN information of the nodes. Requires `pglogical` and `bdr` extensions installed on all databases.
- `all_bdr_nodes`: If `logical_replication_mode` is set to `bdr`, then you can specify only the Initial Connection and let LiveCompare build the connection list based on the current list of active PGD nodes. Default: `off`.
- `max_parallel_workers`: Number of parallel processes to consider. Each process works on a table from the queue. Default: `2`.

> **Important**
>
> Each process keeps N+1 open connections: one to each data connection and another one to the output database.

- `buffer_size`: Number of rows to retrieve from the tables on every data fetch operation. Default: `4096`.
- `log_level`: Verbosity level in the log file. Possible values: `debug`, `info`, `warning`, or `error`. Default: `info`.
- `data_fetch_mode`: Affects how LiveCompare fetches data from the database.
  - `prepared_statements`: Uses prepared statements (a query with `LIMIT`) for data fetch. Only a very small amount of data (`buffer_size = 4096` rows by default) is fetched each time, so it has the smallest impact of all three modes, and for the same reason it's the safer fetch mode. Allows asynchronous data fetch (defined by `parallel_data_fetch`). For the general use case, this fetch method provides good performance, but a performance decrease can be felt for large tables. This is the default and strongly recommended when server load is medium-high.
  - `server_side_cursors_with_hold`: Uses server-side cursors `WITH HOLD` for data fetch. As table data is retrieved in a single transaction, it holds back `xmin` and can cause bloat and replication issues and also prevent `VACUUM` from running well. Also, the `WITH HOLD` clause tells Postgres to materialize the query (workers can hang for a few seconds waiting for the data to materialize), so the whole table data consumes RAM and can be stored on Postgres side disk as temporary files. You can reduce all that impact by using `parallel_chunk_rows` (disabled by default), and improve speed by increasing `buffer_size` a little. Allows asynchronous data fetch (defined by `parallel_data_fetch`). For the general use case, this fetch method doesn't provide any benefits when compared to `prepared_statements`, but for multiple small tables it's faster. However, this mode is recommended only when load is very low, for example, on tests and migration scenarios.
  - `server_side_cursors_without_hold`: Uses server-side cursors `WITHOUT HOLD` for data fetch. As `server_side_cursors_with_hold`, this mode can also hold back `xmin`, thus it potentially can cause bloat, `VACUUM`, and replication issues on Postgres. However, such impact is higher because `WITHOUT HOLD` cursors require an open transaction for the whole comparison session (this requirement will be lifted in later versions). As the snapshot is held for the whole comparison session, comparison results might be helpful depending on your use case. As the query isn't materialized, memory usage and temp file generation remains low. Asynchronous data fetch isn't allowed. In terms of performance, this mode is slower for the general use case, but for large

tables it can be the faster. We recommend it when load on the database is low-medium.

> **Important**
>
> The choice of the right `data_fetch_mode` for the right scenario is very important. Using prepared statements has the smallest footprint on the database server, so it's the safest approach, and it's good for the general use case. Another point is that prepared statements allow LiveCompare to always see the latest version of the rows, which might not happen when using server-side cursors on a busy database. So we recommend using `prepared_statements` for production, high-load servers and either `server_side_cursors_*` setting for testing, migration scenarios, and low-load servers. The best strategy probably mixes `server_side_cursors_without_hold` for very large tables and `prepared_statements` for the remaining tables. The following table shows a comparison of the cost/benefit ratio.

	prepared_statements	server_side_cursors_with_hold	server_side_cursors_without_hold
xmin hold	very low	medium	high
xmin released per	buffer	chunk	whole comparison session
temp files	very low	very high	low
memory	very low	high	low
allows async conns	yes	yes	no
fastest for	general	small tables	large tables
recommended load	high	very low	low-medium

> **Note about Oracle**
>
> For Oracle, the `data_fetch_mode` setting is completely ignored, and data is always fetched from Oracle using a direct query. Data is taken in chunks of `buffer_size` through the client-side cursor.

- `parallel_chunk_rows` : Minimum number of rows required to consider splitting a table into multiple chunks for parallel comparison. A hash is used to fetch data, so workers don't clash with each other. Each table chunk has no more than `parallel_chunk_rows` rows. Setting it to any value <1 disables table splitting. Default: `0` (disabled).

> **Important**
>
> While table splitting can help multiple workers to compare a large table in parallel, performance for each worker can be affected by the hash condition being applied to all rows. Depending on the Postgres configuration (especially with the default of `random_page_cost = 4` , which can be considered too conservative for modern hard drives), the Postgres query planner can incorrectly prefer bitmap heap scans. If the database is running on SSD, disabling bitmap heap scan on LiveCompare can significantly improve the comparison performance. You can do this per connection using the `start_query` setting:

```
start_query = set enable_bitmapscan =
off
```

- `parallel_data_fetch` : Specifies whether data fetch is performed in parallel (that is, using async connections to the databases). Improves performance of multi-way comparison. If any data connections aren't PostgreSQL, then this setting is automatically disabled. It's allowed only when `data_fetch_mode = prepared_statements` or `data_fetch_mode = server_side_cursors_with_hold` . Default: `on` .

- `comparison_algorithm` : Affects how LiveCompare works through table rows to compare data. Using hashes is faster than full-row comparison. It can assume one of the following values:

  - `full_row` : Disables row comparison using hashes. Full comparison, in this case, is performed by comparing the row column by column. Note that for comparisons involving Oracle 10g database, `full_row` is the only valid `comparison_algorithm` setting value.

  - `row_hash` : Enables row comparison using hashes and enables table splitting. Tables are split so each worker compares a maximum of `parallel_chunk_rows` per table. Data row is hashed in PostgreSQL, so the comparison is faster than `full_row` . However, if the hash for a specific row doesn't match, then for that specific row, LiveCompare falls back to the `full_row` algorithm (that is, compare row by row). If any data connection isn't PostgreSQL, then LiveCompare uses a row hash that's defined as the MD5 hash of the concatenated

column values of the row being considered, a *common hash* among the database technologies being compared.

  - `block_hash` : Works the same as `row_hash` , but instead of comparing row by row, LiveCompare builds a *block hash*, that is, a hash of the hashes of all rows in the data buffer that was just fetched (maximum of `buffer_size` rows). Conceptually it works like a two-level Merkle tree. If the block hash matches, then LiveCompare advances the whole block, which is why this comparison algorithm is faster than `row_hash` . If block hash doesn't match, then LiveCompare falls back to `row_hash` and performs the comparison row by row in the buffer to find the divergent rows. This is the default value.

- `min_time_between_heart_beats` : Time in seconds to wait before logging a *heart beat* message to the log. Each worker tracks it separately per round part being compared. Default: 30 seconds.

- `min_time_between_round_saves` : Time in seconds to wait before updating each round state when the comparison algorithm is in progress. A round save can happen only during a heart beat, so `min_time_between_round_saves` must be greater than or equal to `min_time_between_heart_beats` . When the round finishes, LiveCompare always updates the round state for that table. Default: 60 seconds.

  > **Important**
  >
  > If you cancel execution of LiveCompare by pressing **Ctrl-C** and start it again, then LiveCompare resumes the round for that table, starting from the point where the round state was saved.

- `comparison_cost_limit` : If > 0, corresponds to a number of rows each worker processes before taking a nap of `comparison_cost_delay` seconds. Defaults to 0, meaning that each worker processes rows without taking a nap.

- `comparison_cost_delay` : If `comparison_cost_limit > 0` , then this setting specifies how long each worker sleeps. Default: `0.0` .

- `stop_after_time` : Time in seconds after which LiveCompare stop as if you press**Ctrl-C**. You can resume the comparison session that was interrupted, if not finished yet, by passing the session ID as an argument in the command line. Default: `stop_after_time = 0` , which means that automatic interruption is disabled.

- `consensus_mode` : Consensus algorithm used by LiveCompare to determine which data connections are divergent. Possible values are `simple_majority` , `quorum_based` , or `source_of_truth` . If `consensus_mode = source_of_truth` , then `difference_sources_of_truth` must be filled. Default: `simple_majority` .

- `difference_required_quorum` : If `consensus_mode = quorum_based` , then this setting specifies the minimum quorum required to decide which connections are divergent. Must be a number between 0.0 and 1.0. 0.0 means no connection is required, and 1.0 means all connections are required. Both cases are extreme and we don't recommend using them. The default value is 0.5, and we recommend using a value close to that.

- `difference_sources_of_truth` : Comma-separated list of connections names (or node names, if `logical_replication_mode = bdr` and `all_bdr_nodes = on`) to consider as the source of truth. It's used only when `consensus_mode = source_of_truth` . For example: `difference_sources_of_truth = node1,node2` . In this example, either the sections `node1 Connection` and `node2 Connection` must be defined in the `.ini` file or `all_bdr_nodes = on` and only the `Initial Connection` is defined, while `node1` and `node2` must be valid PGD node names.

- `difference_tie_breakers` : Comma-separated list of connection names (or node names, if `logical_replication_mode = bdr` and `all_bdr_nodes = on`) to be considered as tie breakers whenever the consensus algorithm finds a tie situation. For example: `difference_tie_breakers = node1,node2` . In this example, either the sections `node1 Connection` and `node2 Connections` must be defined in the `.ini` file or `all_bdr_nodes = on` and only the `Initial Connection` is defined, while `node1` and `node2` must be valid PGD node names. Default: Don't consider any connection as tie breaker.

- `difference_statements` : Controls the kind of DML statements for LiveCompare to generate. The value of `difference_statements` can be one of:

  - `all` (default)
  - `inserts`
  - `updates`
  - `deletes`

    - `inserts_updates`
    - `inserts_deletes`
    - `updates_deletes`

- `difference_allow_null_updates` : Determines whether commands like `UPDATE SET col = NULL` are allowed in the difference report. Default: `on` .

- `difference_statement_order` : Controls order of DML statements that LiveCompare generates. The value of `difference_statement_order` can be one of:

    - `delete_insert_update`
    - `delete_update_insert` (default)
    - `insert_update_delete`
    - `insert_delete_update`
    - `update_insert_delete`
    - `update_delete_insert`

- `difference_fix_replication_origin` : When working with PGD databases, for difference, LiveCompare creates a specific replication origin if it doesn't exist yet. It then uses the replication origin to create an apply script with DML fixes. The setting `difference_fix_replication_origin` specifies the name of the replication origin used by LiveCompare. If you don't set any value for this setting, then LiveCompare sets `difference_fix_replication_origin = bdr_local_only_origin` . The replication origin that LiveCompare creates isn't dropped to allow verification after the comparison. However, if needed, you can manually drop the replication origin later. Requires `logical_replication_mode = bdr` .

> **Important**
>
> PGD 3.6.18 introduced the new pre-created `bdr_local_only_origin` replication origin to use for applying local-only transactions. So if LiveCompare is connected to PGD 3.6.18, it doesn't create this replication origin, and we recommend you don't try to drop this replication origin.

- `difference_fix_start_query` : Arbitrary query that's executed at the beginning of the apply script generated by LiveCompare. Additionally, if a PGD comparison is being performed and the `difference_fix_start_query` is empty, then LiveCompare also automatically does the following:

    - If the divergent connection is PGD 3.6.7, adds `SET LOCAL bdr.xact_replication = off;`
    - Adds commands that set up transaction to use the replication origin specified in `difference_fix_replication_origin`

- `show_progress_bars` : Determines whether to show progress bars in the console output. Disabling this setting might be useful for batch executions. Default: `on` .

- `output_schema` : In the output connection, the schema where the comparison report tables are created. Default: `livecompare` .

- `hash_column_name` : Every data fetch contains a specific column that's the hash of all actual columns in the row. This setting specifies the name of this column. Default: `livecompare_hash` .

- `rownumber_column_name` : Some fetches need to use the `row_number()` function value inside a query column. This setting specifies the name of this column. Default: `livecompare_rownumber` .

- `fetch_row_origin` : When this setting is enabled, LiveCompare fetches the origin name for each divergent row, which might be useful for debugging purposes. To be enabled, requires `logical_replication_mode` set to `pglogical` or `bdr` . Default: `off` .

- `column_intersection` : When this setting is enabled, for a given table that's being compared, LiveCompare works only on the intersection of columns from the table on all connections, ignoring extra columns that might exist on any of the connections. When this setting is disabled, LiveCompare checks if columns are equivalent on the table on all connections and aborts the comparison of the table if there are any column mismatches. Default: `off` .

> **Important**

> If a table has PK, then the PK columns aren't allowed to be different, even if `column_intersection = on`.

- `ignore_nullable`: For a specific table comparison, if LiveCompare is using a comparison key different from the primary key, then LiveCompare requires all columns to be `NOT NULL` if `ignore_nullable` is enabled (default). You can override that behavior by setting `ignore_nullable = off`, which allows LiveCompare to consider null-able columns in the comparison, which in some corner cases can produce false positives.

- `check_uniqueness_enforcement`: If LiveCompare is using a user-defined comparison key or using all columns in the table as a comparison key, then LiveCompare checks for table uniqueness on the comparison key if setting `check_uniqueness_enforcement` is enabled (default).

- `oracle_ignore_unsortable`: When enabled, tells LiveCompare to ignore columns with Oracle unsortable data types (BLOB, CLOB, NCLOB, BFILE) if column isn't part of the table PK. If enabling this setting, we recommend also enabling `column_intersection`.

- `oracle_user_tables_only`: When enabled, tells LiveCompare to fetch table metadata only from the Oracle logged-in user. This approach is faster because it reads, for example, from `sys.user_tables` and `sys.user_tab_columns` instead of `sys.all_tables` and `sys.all_tab_columns`. Default: `off`.

- `oracle_fetch_fk_metadata`: When enabled, tells LiveCompare to fetch foreign-key metadata, which can be a slow operation. Overrides the value of the setting `fetch_fk_metadata` on the Oracle connection. Default: `off`.

- `schema_qualified_table_names`: Table names are treated as schema qualified when this setting is enabled. Disabling it allows comparing tables without using schema-qualified table names. On Oracle x Postgres comparisons, it requires also enabling `oracle_user_tables_only`. On Postgres x Postgres, it allows for comparisons of tables that are under different schemas, even in the same database. Also, when `schema_qualified_table_names` is enabled, `Table Filter -> tables`, `Row Filter`, and `Column Filter` allow table name without the schema name. Default: `on`.

- `force_collate`: When set to a value other than `off` and to a valid collation name, forces the specified collation name in `ORDER BY` operations in all Postgres databases being compared. Useful when comparing Postgres databases with different collation or when comparing Oracle and Postgres databases. (In this case, set `force_collate = C`.) Assumes value `C` if comparing mixed technologies (like Oracle versus PostgreSQL) and no collation is specified. Default: `off`.

- `work_directory`: Path to the `LiveCompare` working directory. The session folder containing output files is created in this directory. Default: `.` (current directory).

- `abort_on_setup_error`: When enabled, if LiveCompare encounters any error when trying to set up a table comparison round, the whole comparison session is aborted. Default: `off`.

> **Important**
>
> Setting `abort_on_setup_error` is considered only during `compare` mode. In `recheck` mode, LiveCompare always aborts at the first error in setup.

- `custom_dollar_quoting_delimiter`: When LiveCompare finds differences, it outputs the DML using dollar quoting on strings. The default behavior is to create a random string to compose it. If you want by any means to use a custom one, you can set this parameter as the delimiter to use. You need to set only the constant, not the `$` symbols around the constant. Default: `off`, which means LiveCompare uses an `md5` hash of the word `LiveCompare`.

- `session_replication_role_replica`: When enabled, LiveCompare uses the `session_replication_role` PostgreSQL setting as `replica` in the output apply scripts. That's useful if you want to prevent firing triggers and rules while applying DML in the nodes with divergences. Enabling it requires a PostgreSQL superuser. Otherwise, it has no effect. Default: `off`.

- `split_updates`: When enabled, LiveCompare splits `UPDATE` divergences. That is, instead of generating an `UPDATE` DML, it generates corresponding `DELETE` and `INSERT` in the apply script. Default: `off`.

- `float_point_round`: An integer to specify decimal digits that LiveCompare rounds when comparing float-point values coming from the database. Default: `-1`, which disables float-point rounding.

Initial Connection

The initial connection is used only when `logical_replication_mode` is set to `pglogical` or `bdr`. If you set data connections to use only the `node_name` setting, it's used when the program starts to fetch DSN from node names.

- `technology`: RDBMS technology. Currently the only possible value is `postgresql`.
- `dsn`: PostgreSQL connection string. If `dsn` is set, then `host`, `port`, `dbname`, and `user` are ignored. The `dsn` setting can also have all other parameter key words allowed by libpq.
- `host`: Server address. Leave empty to use the Unix socket connection.
- `port`: Port. Default: `5432`.
- `dbname`: Database name. Default: `postgres`.
- `user`: Database user. Default: `postgres`.
- `application_name`. Application name. Can be used even if you set `dsn` instead of all other connection information. Default: `livecompare_initial`.

Output Connection

The output connection specifies where LiveCompare creates the comparison report tables.

- `technology`: RDBMS technology. Currently the only possible value is `postgresql`.
- `dsn`: PostgreSQL connection string. If `dsn` is set, then `host`, `port`, `dbname`, and `user` are ignored. The `dsn` setting can also have all other parameter key words allowed by libpq.
- `host`: Server address. Leave empty to use the Unix socket connection.
- `port`: Port. Default: `5432`.
- `dbname`: Database name. Default: `postgres`.
- `user`: Database user. Default: `postgres`.
- `application_name`. Application name. Can be used even if you set `dsn` instead of all other connection information. Default: `livecompare_output`.

Data Connection

A data connection is a connection section similar to `Initial Connection` and `Output Connection`, but LiveCompare effectively fetches and compares data on the data connections.

Similar to the `Initial Connection` and `Output Connection`, a data connection is defined in a named section. The section name is of the form `<Name> Connection`, with `<Name>` being any single-word string starting with an alphabetic character. In this case, whatever you use as `Name` is called the *connection ID* of the data connection. Each data connection must also have a unique connection ID in the list of data connections.

If `logical_replication_mode = bdr` and `all_bdr_nodes = on`, then you don't need to specify any data connection. LiveCompare builds the data connection list by fetching PGD metadata from the `Initial Connection`.

- `technology`: RDBMS technology. Currently possible values are `postgresql` or `oracle`.
- `node_name`: Name of the node in the cluster. Requires `logical_replication_mode` set to `pglogical` or `bdr` and also requires that the `Initial Connection` is filled. If `node_name` is set, then `dsn`, `host`, `port`, `dbname`, and `user` settings are all ignored.
- `dsn`: PostgreSQL connection string. If `dsn` is set, then `host`, `port`, `dbname`, and `user` are ignored. The `dsn` setting can also have all other parameter key words allowed by libpq.
- `host`: Server address. Leave empty to use the Unix socket connection.
- `port`: Port. Default: `5432`.
- `dbname`: Database name. Default: `postgres`.
- `service`: Service name, used in Oracle connections. Default: `XE`.
- `user`: Database user. Default: `postgres`.
- `password`: Plain text password. We don't recommend using this. However, it might be required in some legacy connections.

- `application_name` . Application name. Can be used even if you set `dsn` or `node_name` instead of all other connection information. Default: `livecompare_<Connection ID>` .
- `start_query` : Arbitrary query that's executed each time a connection to a database is open.
- `fetch_fk_metadata` : Specifies whether LiveCompare gathers metadata about foreign keys on the connection. Default: `on` .

Table Filter

If omitted or left empty, this section from the `.ini` file means that LiveCompare executes against all tables in the first database.

If you want LiveCompare to execute against a specific set of tables, there are different ways to specify this:

- `publications` : You can filter specific publications, and LiveCompare uses only the tables associated with those publications. You can use the variable `publication_name` to build the conditional expression, for example:

  ```
  publications = publication_name =
  'livepub'
  ```

  Requires `logical_replication_mode = native` .

- `replication_sets` : When using pglogical or PGD, you can filter specific replication sets, and LiveCompare works only on the tables associated with those replication sets. You can use the variable `set_name` to build the conditional expression, for example:

  ```
  replication_sets = set_name in ('default',
  'bdrgroup')
  ```

  Requires `logical_replication_mode = pglogical` or `logical_replication_mode = bdr` .

- `schemas` : You can filter specific schemas, and LiveCompare works only on the tables that belong to those schemas. You can use the variable `schema_name` to build the conditional expression, for example:

  ```
  schemas = schema_name !=
  'badschema'
  ```

- `tables` : The variable `table_name` can help you build a conditional expression to filter only the tables you want LiveCompare to work on, for example:

  ```
  tables = table_name not like
  '%%account'
  ```

  In any conditional expression, escape the `%` character as `%%` .

The table name must be schema-qualified, unless `schema_qualified_table_names` is disabled. For example, you can filter only a specific list of tables:

```
tables = table_name in ('myschema1.mytable1', 'myschema2.mytable2')
```

If you disable the general setting `schema_qualified_table_names` , then you must also set an appropriate `search_path` for Postgres in the connection `start_query` setting, for example:

```
[General Setting]
...
schema_qualified_table_names = off

[My Connection]
```

```
...
start_query = SET search_path TO myschema1, myschema2

[Table Filter]
tables = table_name in ('mytable1', 'mytable2')
```

> **Important**
>
> If two or more schemas that were set on `search_path` contain a table with the same name, just the first one found is considered in the comparison.

The `Table Filter` section can have a mix of `publications`, `replication_sets`, `schemas`, and `tables` filters. LiveCompare considers the set of tables that are in the intersection of all filters you specified. For example:

```
[Table Filter]
publications = publication_name =
'livepub'
replication_sets = set_name in ('default',
'bdrgroup')
schemas = schema_name !=
'badschema'
tables = table_name not like
'%%account'
```

The table filter is applied in the first database to build the table list. If a table exists in the first database and is being considered in the filter, but it doesn't exist in any other database, then you something like this is added to the logs, and the comparison for that specific table is skipped:

```
2019-06-17 11:52:41,403 - ERROR - live_table.py - 55 - GetMetaData - P1: livecompare_second_1: Table
public.test does not exist
2019-06-17 11:52:41,410 - ERROR - live_round.py - 201 - Initialize - P1: Table public.test does not exist
on second connection. Aborting comparison
```

Similarly, if a table exists in any other database but doesn't exist in the first database, then it isn't considered in the comparison, even if you didn't apply any table filter.

A comparison for a specific table is also skipped if the table column names aren't exactly the same (unless `column_intersection` is enabled), and in the same order. An appropriate message is added to the log file as well.

Currently LiveCompare doesn't check if data types or constraints are the same on both tables.

> **Important**
>
> `conflicts` mode doesn't make use of the table filter.

Row Filter

In this section, you can apply a row-level filter to any table, so LiveCompare works only on the rows that satisfy the row filter.

You can write a list of tables under this section, one table per line. All table names must be schema qualified unless `schema_qualified_table_names` is disabled. For example:

```
[Row Filter]
public.table1 = id =
10
```

```
public.table2 = logdate >= '2000-01-
01'
```

In this case, for the table `public.table1`, LiveCompare works only in the rows that satisfy the clause `id = 10`. For the table `public.table2`, only rows that satisfy `logdate >= '2000-01-01'` are considered in the comparison.

If you disable the general setting `schema_qualified_table_names`, then you must also set an appropriate `search_path` for Postgres in the connection `start_query` setting, for example:

```
[General Setting]
...
schema_qualified_table_names = off

[My Connection]
...
start_query = SET search_path TO public

[Row Filter]
table1 = id = 10
table2 = logdate >= '2000-01-01'
```

Any kind of SQL condition (same as you put in the `WHERE` clause) is accepted in the same line as the table row filter. For example, if you have a large table and want to compare only a specific number of IDs, you can create a temporary table with all the IDs. Then you can use an `IN` clause to emulate a `JOIN`, like this:

```
[Row Filter]
public.large_table = id IN (SELECT id2 FROM temp_table)
```

If a row filter is written incorrectly, then LiveCompare tries to apply the filter but fails. So the comparison for this specific table is skipped, and an exception is written to the log file.

If a table is listed in the `Row Filter` section but somehow got filtered out by the `Table Filter`, then the row filter for this table is silently ignored.

> **Important**
>
> `conflicts` mode doesn't make use of the row filter.

Using current timestamp in Row Filter

The `Row Filter` is applied differently depending on the `data_fetch_mode`:

- On Postgres, setting `data_fetch_mode` to `server_side_cursors_with_hold` or `server_side_cursors_without_hold` causes the `Row Filter` to be applied only at the beginning of the table comparison, when the query is executed. This means that using a server-side cursor to fetch data ensures the data is seen as a snapshot of how it was beginning of the comparison.
- On Postgres, setting `data_fetch_mode` to `prepared_statements` (the default) includes the `Row Filter` in the prepared query, which is then executed at every data buffer that's fetched. This means that, if the query uses `now()`, `CURRENT_TIMESTAMP`, or `SYSDATE` (on EDB Postgres Advanced Server) on the `Row Filter`, then when the prepared statement executes, Postgres reevaluates the current timestamp.

So, suppose you're using `now()`, `CURRENT_TIMESTAMP`, or `SYSDATE` on the `Row Filter`, for example:

```
[Row Filter]
public.table3 = logdate < CURRENT_TIMESTAMP
```

In this case, you must also use a server-side cursor to ensure the current timestamp is evaluated only at the beginning of the queries. In other words,

`data_fetch_mode` must be set to a value different from `prepared_statements`.

On Oracle, the `data_fetch_mode` setting is ignored, and the query is executed at the beginning. Then data is fetched by way of the client-side cursor. This approach ensures data is seen as a snapshot of how it was at the beginning of the comparison. This is a client-side cursor, but the behavior is similar to using a server-side cursor in Postgres.

Column Filter

In this section, you can apply a column-level filter to any table, so LiveCompare works only on the columns that aren't part of the column filter.

You can write a list of tables under this section, one table per line. All table names must be schema qualified unless `schema_qualified_table_names` is disabled. For example, suppose that both `public.table1` and `public.table2` have the columns `column1`, `column2`, `column3`, `column4`, and `column5`:

```
[Column Filter]
public.table1 = column1,
column3
public.table2 = column1,
column5
```

In this case, for the table `public.table1`, LiveCompare works only in the columns `column2`, `column4`, and `column5`, filtering out `column1` and `column3`. For the table `public.table2`, only the columns `column2`, `column3`, and `column4` are considered in the comparison, filtering out `column1` and `column5`.

If you disable the general setting `schema_qualified_table_names`, then you must also set an appropriate `search_path` for Postgres in the connection `start_query` setting, for example:

```
[General Setting]
...
schema_qualified_table_names = off

[My Connection]
...
start_query = SET search_path TO public

[Column Filter]
table1 = column1, column3
table2 = column1, column5
```

If absent column names are given in the column filter, that is, the column doesn't exist in the given table, then LiveCompare logs a message about the missing columns and ignores them. It uses just the valid ones, if any.

If a table is listed in the `Column Filter` section but somehow got filtered out by the `Table Filter`, then the column filter for this table is silently ignored.

> **Important**
>
> If a column specified in a `Column Filter` is part of the table PK, then it isn't ignored in the comparison. LiveCompare logs that and ignores the filter of such a column.

> **Important**
>
> `conflicts` mode doesn't make use of the column filter.

Comparison Key

> **New feature**
>
> LiveCompare comparison key support is available in LiveCompare version 2.0 and later.

Similar to the `Column Filter`, in this section you can also specify a list of columns per table. These columns are considered as a comparison key for the specific table, even if the table has a primary key or `UNIQUE` constraint.

For example:

```
[Comparison Key]
public.table1 = col_a,
col_b
public.table2 = c1,
c2
```

In this example, for table `public.table1`, the comparison key is columns `col_a` and `col_b`. For table `public.table2`, columns `c1` and `c2` are considered as a comparison key.

The same behavior about missing columns or filtered out or missing tables that are explained inColumn Filter, also apply to the comparison key. Similarly, the `Comparison Key` section is ignored in `conflicts` mode.

Conflicts Filter

In this section, you can specify a filter to use in `--conflicts` mode while fetching conflicts from PGD nodes. You can build any SQL conditional expression and use these fields in the expression:

- `origin_node`: The upstream node of the subscription.
- `target_node`: The downstream node of the subscription.
- `local_time`: The timestamp when the conflict occurred in the node.
- `conflict_type`: The type of conflict.
- `conflict_resolution`: The resolution that was applied.
- `nspname`: Schema name of the involved relation.
- `relname`: Relation name of the involved relation.

You must use the `conflicts` attribute under the section. For example:

```
[Conflicts Filter]
conflicts = conflict_type = 'update_missing' AND nspname = 'my_schema'
```

If you add this piece of configuration to your `.ini` file, LiveCompare fetches only conflicts that are of type `update_missing` and related to tables under the schema `my_schema` while querying for conflicts in each of the PGD nodes.

> **Important**
>
> This section is exclusively for `--conflicts` mode.

10 Licenses

TQDM

`tqdm` is a product of collaborative work. Unless otherwise stated, all authors (see commit logs) retain copyright for their respective work, and release the work under the MIT licence (text below).

Exceptions or notable authors are listed below in reverse chronological order:

- files: * MPLv2.0 2015-2020 (c) Casper da Costa-Luis casperdcl.
- files: tqdm/_tqdm.py MIT 2016 (c) PR #96 on behalf of Google Inc.
- files: tqdm/_tqdm.py setup.py README.rst MANIFEST.in .gitignore MIT 2013 (c) Noam Yorav-Raphael, original author.

Mozilla Public License (MPL) v. 2.0 - Exhibit A

This Source Code Form is subject to the terms of the Mozilla Public License, v. 2.0. If a copy of the MPL wasn't distributed with this file, you can obtain one at https://mozilla.org/MPL/2.0/.

MIT License (MIT)

Copyright (c) 2013 noamraph

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

cx_Oracle

LICENSE AGREEMENT FOR CX_ORACLE

Copyright © 2016, 2020, Oracle and/or its affiliates. All rights reserved.

Copyright © 2007-2015, Anthony Tuininga. All rights reserved.

Copyright © 2001-2007, Computronix (Canada) Ltd., Edmonton, Alberta, Canada. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

Redistributions of source code must retain the above copyright notice, this list of conditions, and the disclaimer that follows. Redistributions in binary form must reproduce the above copyright notice, this list of conditions, and the following disclaimer in the documentation and/or other materials provided with the distribution. Neither the names of the copyright holders nor the names of any contributors may be used to endorse or promote products derived from this software without specific prior written permission. DISCLAIMER: THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS *AS IS* AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF

THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Computronix ® is a registered trademark of Computronix (Canada) Ltd.

Apache license

```
                                 Apache License
                           Version 2.0, January 2004
                        http://www.apache.org/licenses/
```

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

   "Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

   "You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

   "Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

   "Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

   "Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

   "Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

   "Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

   "Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

   (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

   You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

   END OF TERMS AND CONDITIONS

Psycopg2

psycopg2 is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

psycopg2 is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

In addition, as a special exception, the copyright holders give permission to link this program with the OpenSSL library (or with modified versions of OpenSSL that use the same license as OpenSSL), and distribute linked combinations including the two.

You must obey the GNU Lesser General Public License in all respects for all of the code used other than OpenSSL. If you modify file(s) with this exception, you may extend this exception to your version of the file(s), but you are not obligated to do so. If you do not wish to do so, delete this exception statement from your version. If you delete this exception statement from all source files in the program, then also delete it here.

You should have received a copy of the GNU Lesser General Public License along with psycopg2 (see the doc/ directory.) If not, see https://www.gnu.org/licenses/.

Alternative licenses

The following BSD-like license applies (at your option) to the files following the pattern `psycopg/adapter*.{h,c}` and `psycopg/microprotocol*.{h,c}`:

Permission is granted to anyone to use this software for any purpose, including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.

2. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.

3. This notice may not be removed or altered from any source distribution.

Tabulate

Copyright (c) 2011-2020 Sergey Astanin and contributors

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE. © 2020 GitHub, Inc.

OmniDB

MIT License

Portions Copyright (c) 2015-2019, The OmniDB Team Portions Copyright (c) 2017-2019, 2ndQuadrant Limited