LiveCompare

Version 1

Built at 2024-08-21T17:33:41

1 LiveCompare

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

By default, the comparison set will include 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 within a table.

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

Each table comparison operation is called a "comparison round". If the table is too big, LiveCompare will split the table into multiple comparison rounds that will also be executed in parallel, alongside with 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 started are ignored. LiveCompare will sort the PK columns in order to get min and max PK from each table. For each PK column which is unsortable, LiveCompare will cast it's content to `string`. In PostgreSQL that's achieved by using `::text` and in Oracle by using `to_char`.

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

Any differences found by the comparison algorithm can be manually re-checked by the user at a later convenient time. This is recommended to be done to allow a replication consistency check. Upon the difference re-check, maybe replication caught up on that specific row and the difference does not exist anymore, so the difference is removed, otherwise it is marked as permanent.

At the end of the execution the program generates a DML script so the user can review it, and fix differences one by one, or simply apply the entire DML script so all permanent differences are fixed.

LiveCompare can be potentially used to ensure logical data integrity at 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 was not properly fenced and the application wrote data into it, it is possible to use LiveCompare to know exactly which data is present in the old primary and is not present in the new primary. If desired, the DBA can use the DML script that LiveCompare generates to apply those data into the new primary;
- Logical replication. Three kind of logical replication technologies are supported: Postgres native logical replication, pglogical and BDR.

Comparison Performance

LiveCompare has been optimized for use on production systems and has various parameters for tuning, described later. 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.

The use case above can be considered a general use case. For low-load, testing, migration and other specific scenarios, it might be possible to improve speed by changing the `data_fetch_mode` setting to use server-side cursors. Each kind of server side cursors, in our experiments, provides an increase in performance on use cases involving either small or large tables.

Security Considerations for the User

When `logical_replication_mode = bdr` , LiveCompare requires a user that has been granted the `bdr_superuser` role. When `logical_replication_mode = pglogical` , LiveCompare requires a user that has been granted the `pglogical_superuser` role.

To apply the DML scripts in BDR, then all divergent connections (potentially all data connections) require a user that has been granted the `bdr_superuser` in order to disable `bdr.xact_replication` .

If BDR is being used, LiveCompare will associate all fixed rows with a replication origin called `bdr_local_only_origin` . LiveCompare will also apply the DML with the transaction datetime far in the past, so if there are any BDR 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 will change role to the owner of the table in order 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 have ability to switch role to the table owner.

2 Release notes

The LiveCompare documentation describes the latest version of LiveCompare 1 including minor releases and patches. The release notes in this section provide information on what is new in each release.

Version	Release Date
1.18.1	Dec 14 2021
1.18.0	Dec 1 2021
1.17.0	Oct 1 2021

2.1 Version 1.18.1

LiveCompare 1.18.1 includes the following bug fix:

Type	Description	ID
Bug fix	Fixed an issue where the internal round part column name setting was not being considered in asynchronous connections.	LIV-95

2.2 Version 1.18.0

LiveCompare 1.18.0 includes the following new features, enhancements, and bug fixes:

Type	Description	ID

Type	Description	ID
Enhancement	Implemented a new QueryBlock data fetch method for Oracle. The algorithm keeps fetching rows from the same socket, which is more efficient as Oracle doesn't allow inequality operators involving Row Value Expressions, which were previously required to fetch rows past the last processed multi-column PK value. This significantly improves performance on Oracle versus Postgres comparisons. As PostgreSQL allows inequality operators involving Row Value Expressions, LiveCompare uses the same algorithm using prepared statements to not cause an impact in terms of bloat and holding back the `xmin` horizon. So Postgres versus Postgres performance is not affected by this implementation.	LIV-83, RT75644, RT75675, RT75332
Enhancement	Removed the need to reset the buffers when using `comparison_algorithm = block_hash`. Now when a block with mismatching hashes is found, LiveCompare correctly switches to `comparison_algorithm = row_hash` until all rows within the mismatching block (already fetched) are compared individually, and then switches back to `comparison_algorithm = block_hash` to fetch the next buffer. This improves performance on Oracle versus Postgres and also Postgres versus Postgres comparisons. For more information, see Common hash.	LIV-85, LIV-86, RT75644, RT75675, RT75332
Enhancement	Now LiveCompare is able to split large tables even on Oracle versus Postgres comparisons, if `comparison_algorithm` is set to `block_hash` or `row_hash` (currently requires Oracle 12 and newer). Previously it was possible only on Postgres versus Postgres, because the hashing is done on the database side. As Oracle and Postgres don't share a common hashing function, we converted the common row hash that's based on MD5 into an integer hash to be able to split table data into multiple workers. On Oracle, due to performance reasons it was not possible to do on the database side, so we calculate this hash on the Python side. The cost of parallelism for Oracle versus Postgres table splitting is still not the same as on Postgres versus Postgres, so on Oracle versus Postgres we recommend to set `parallel_chunk_rows` higher than the default of 10 Million. For example, set to 100 Million or even higher, so the algorithm will split only really large tables. For more information, see Common hash.	LIV-87
Enhancement	Support for EDB Extended PostgreSQL 14 and EDB Advanced PostgreSQL 14.	LIV-71, LIV-72

Type	Description	ID
Enhancement	Collecting the Oracle `rowid` which can be helpful to inspect divergences.	LIV-90, RT75644, RT75675
Enhancement	Improved logging for the LiveRound comparison algorithm, which can be helpful to inspect divergences.	LIV-91, RT75644, RT75675
Bug fix	Fixed the issue where `oracle_fetch_fk_metadata` was not being honored.	LIV-92
Bug fix	Fixed an issue where, if when using `log_level = debug` on log-heavy use cases, LiveCompare could die with an unhandled exception because the logging queue was full. On Unix systems, the maximum size of any queue is 32676. The problem happened because the base Python implementation asynchronously enqueued the logging records. The solution was to override the Python implementation to synchronously enqueue the logging records, which can have a performance impact when `log_level = debug` and log-heavy use cases.	LIV-93

2.3 Version 1.17.0

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

Type	Description	ID
Enhancement	Added support to PostgreSQL 14.	LIV-66
Enhancement	Added support to Debian 11.	LIV-70
Bug fix	Now `--recheck` always requires a session ID.	LIV-76
Bug fix	Minor fixes to the documentation.	LIV-26, LIV-68, LIV-80

3 Requirements

LiveCompare requires:

- Python >= 3.6 and <= 3.8
- PostgreSQL / EDB Postgres Extended 9.5+ / EPAS 11+ (on the output connection)
- PostgreSQL / EDB Postgres Extended 9.4+ / EPAS 11+ or Oracle 10g+ (on the data connections being compared)

LiveCompare requires at least Debian 10, Ubuntu 16.04, or CentOS/RHEL 7.

LiveCompare can be installed from the EnterpriseDB `products/livecompare` repository. More details can be found in:

https://techsupport.enterprisedb.com/customer_portal/sw/livecompare/

LiveCompare installs on top of:

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

On CentOS/RHEL distributions, LiveCompare also requires the EPEL repository. More details can be found in:

https://fedoraproject.org/wiki/EPEL

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

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

LiveCompare with TPAexec

The following sample config for `TPAexec` can be used 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:
    - 2ndq-livecompare
  use_volatile_subscriptions: true

locations:
- Name: main

instance_defaults:
  image: tpa/redhat
  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
```

More details about TPAexec can be found in:

https://techsupport.enterprisedb.com/customer_portal/sw/tpa/

4 Supported Technologies

LiveCompare is able to connect to and compare data from a list of technologies including PostgreSQL, BDR and Oracle.

In LiveCompare there are 3 kinds of connections:

- **Initial** (optional): Used to fetch metadata about pglogical or BDR connections. Required if data connections are pglogical or BDR, and if `replication_sets` or `node_name` settings are used. Requires `logical_replication_mode = pglogical` or `logical_replication_mode = bdr`. It is required to be a pglogical- or BDR-enabled database.
- **Data**: The actual database connection that the tool will connect to perform data comparison. The first connection in the list is used to solve `Table Filter` and `Row Filter`, and is also used in conjunction with the Initial connection to gather information about BDR nodes. If `logical_replication_mode = bdr` and `all_bdr_nodes = on`, then LiveCompare will consider all BDR nodes that are part of the same BDR cluster as the `Initial Connection`. In this case it is not necessary 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 will create a schema called `livecompare`, some tables and views. This is required to keep progress and reporting data about comparison sessions. It is required to be a PostgreSQL or 2ndQPostgres connection.

Below you can find about versions and details about supported technologies and in which context they can be used in LiveCompare.

Technology	Versions	Connections
PostgreSQL	10, 11, 12 and 13	Data and/or Output
EDB PostgreSQL Extended	10, 11, 12 and 13	Data and/or Output
EDB PostgreSQL Advanced (EPAS)	11, 12 and 13	Data and/or Output
pglogical	2 and 3	Initial and/or Data
BDR	1, 2 and 3	Initial and/or Data
Oracle	10g, 11g, 12c and 18c	A single Data connection

PgBouncer Support

LiveCompare can be used against nodes through PgBouncer, but only if using `pool_mode=session` because LiveCompare uses prepared statements on PostgreSQL, and it would not be possible if `pool_mode` were either `transaction` or `statement`.

5 Command-line Usage

Compare mode

Copy any `/etc/2ndq-livecompare/template*.ini` to use in your project and adjust as necessary (see the section `Settings` below).

```
cp /etc/2ndq-livecompare/template_basic.ini my_project.ini

2ndq-livecompare my_project.ini
```

During the execution of LiveCompare, you will see `N+1` progress bars, `N` being the number of processes (you can specify the number of processes in the settings). The first progress bar shows overall execution while 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 may be the ID of the comparison round followed by the current table chunk (`p1/1` means the table was not split). If the status says `setup`, it 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

If a table has more rows than the `parallel_chunk_rows` setting (see more details below), then a hash function will be used to determine which job will consider each row. This can slow down the comparison individually, but the comparison as a whole may benefit from parallelism for the given table.

While the program is executing, you can cancel it at any time by pressing `Ctrl-c`. You will see a message like this:

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

2ndq-livecompare my_project.ini 6
```

Important: If LiveCompare is running in background or running in another shell, you can still softly stop it. It will keep the `PID` of the master process inside the session folder (in the example `lc_session_6`), 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:

```
2ndq-livecompare my_project.ini 6
```

When the program ends, if it found no inconsistencies, you will see an output like this:

```
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.
```

But if any inconsistencies were found, the output will look 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';
```

Re-check mode

In a BDR environment, any divergence that BDR finds can be later non-existing 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, users can manually execute a re-check only on the differences that were previously found. This execution mode is called "recheck" and can be executed like this:

```
 2ndq-livecompare my_project.ini 6 --recheck
```

In this mode, LiveCompare will generate separate recheck logs and update all reports that are already existing in the `lc_session_X` directory.

Important: If resuming a `compare` or executing under `recheck`, LiveCompare will check if the settings and connections attributes are the same as when the session was created. If any divergence found, it will quit the execution with proper message.

Conflicts mode

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

```
2ndq-livecompare my_project.ini --conflicts
```

For more details about the `conflicts` mode, check BDR Support chapter.

6 Advanced Usage

After the end of execution of LiveCompare, you will notice it created 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` : shows a list of all tables that were processed, and 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 also 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` : if there are any differences, this file shows useful information about each difference. This file is not generated if there are no differences.

For example, the difference list could be like this:

```
+-------------------+-------------------------+-----------------+---------------------+
| 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 also execute the following query against the output database:

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

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

```
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 will show a DML command to be applied on the **first** database, to make the **first** database consistent all other databases. For example, for the differences above, this script could be:

```
 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 automatically. In order to fix the inconsistencies in the **first** database, you can simply execute the script in the **first** database.

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

Which differences to fix

LiveCompare is able to identify and provide fixes for the following differences:

- A row exists in the majority of the data connections. The fix will be an `INSERT` on the divergent databases;
- A row does not exist in the majority of the data connections. The fix will be a `DELETE` on the divergent databases;
- A row exists in all databases, but some column values mismatch. The fix will be an `UPDATE` on the divergent databases.

By default `difference_statements = all` , which means that LiveCompare will try to apply all 3 DML types (`INSERT` , `UPDATE` and `DELETE`) for each difference it finds. But it is possible to specify which type of DML LiveCompare should consider when providing difference fixes, by changing the value of the setting `difference_statements` , which can be:

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

When `difference_statements` has the values `all` , `updates` , `inserts_updates` or `updates_deletes` , then it is possible to tell LiveCompare to ignore any `UPDATE` s that would set `NULL` to a column.

Difference log

Table `difference_log` stores all information about differences every time LiveCompare checked them. Users can run LiveCompare in re-check mode multiple times, so this table shows how the difference has evolved over the time window where LiveCompare was re-checking it.

- **Detected (D)**: The difference was just detected. In re-check and fix modes, LiveCompare will mark all Permanent and Tie differences as Detected in order to re-check them.

- **Permanent (P)**: After having re-checked the difference, if data is still divergent, LiveCompare marks the difference as **Permanent**.

- **Tie (T)**: Same as Permanent, but there is not enough consensus to determine which connections are the majority.

- **Absent (A)**: If upon a re-check LiveCompare finds that the difference does not exist anymore (the row is now consistent between both databases), then LiveCompare marks the difference as **Absent**.

- **Volatile (V)**: If upon a re-check `xmin` has changed on an inconsistent row, then LiveCompare marks the difference as **Volatile**.

- **Ignored (I)**: Users can stop difference re-check 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 BDR Support

LiveCompare can be used against BDR nodes, as well as non-BDR nodes.

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

For example, consider you are able to connect to any node in the BDR cluster. Let's call this `Initial Connection`. By initially connection to this node, LiveCompare is able to check BDR metadata and retrieve connection information from all other nodes.

Now consider you want to compare 3 BDR nodes. As LiveCompare is able to connect to any node starting from the `Initial Connection`, you do not need to define `dsn` or any connection information for the data connections. You just need to define `node_name`. LiveCompare searches in BDR metadata about the connection information for that node, and then connects to the node.

Please note that, for LiveCompare to be able to connect to all other nodes by fetching BDR metadata, it is required that LiveCompare is able to connect to them using the same DSN from BDR view `bdr.node_summary`, field `interface_connstr`. In this case it is recommended to run LiveCompare on the same machine as the `Initial Connection`, as `postgres` user. If that's not possible, then please define the `dsn` attribute in all data connections.

You can also specify replication sets as table filters. LiveCompare will use BDR metadata to build the table list, considering only tables that belong to the replication set(s) you defined in the `replication_sets` setting.

For example, you can create an `.ini` file to compare 3 BDR nodes:

```
[General Settings]
logical_replication_mode =
bdr
max_parallel_workers = 4
parallel_chunk_rows = 1000000

[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'
```

It is also possible to tell LiveCompare to compare all active nodes in the BDR cluster. For that purpose just do the following:

- In `General Settings`, enable `all_bdr_nodes = on`;
- Specify an `Initial Connection`;
- Additional data connections are not required.

For example:

```
 [General Settings]
logical_replication_mode =
bdr
max_parallel_workers = 4
parallel_chunk_rows = 1000000
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` to fetch the list of all BDR nodes. Additional data connections are not required; although if set, will be appended to the list of data connections. For example, it would be possible to compare a whole BDR cluster against a single Postgres connection, useful in migration projects:

```
 [General Settings]
logical_replication_mode =
bdr
max_parallel_workers = 4
parallel_chunk_rows = 1000000
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'
```

Settings `node_name` and `replication_sets` are supported for the following technologies:

- BDR 1, 2 and 3;
- pglogical 2 and 3.

Please note that to enable pglogical metadata fetch instead of BDR, just set `logical_replication_mode = pglogical` instead of `logical_replication_mode = bdr`.

BDR Witness nodes

Using replication sets in BDR, it's possible to configure specific tables to be included in the BDR replication, and also specify which nodes should receive data from such tables, by configuring the node to subscribe to the replication set the table belongs to. This allows for different architectures such as BDR Sharding and the use of BDR Witness nodes.

A BDR Witness is a regular BDR node which doesn't replicate any DML from other nodes. The purpose of the Witness is to provide quorum in Raft Consensus voting (for more details on the BDR Witness node, check BDR documentation). Depending on how replication sets were configured, the Witness may or may not replicate DDL. Which means that there are 2 types of BDR Witnesses:

- A completely empty node, without any data nor tables; or
- A node that replicates DDL from other nodes, hence having empty tables.

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

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

In the second case, on the other hand, the table exists on the BDR Witness. However, it would not be 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;
- The replication sets that the table is associated with;
- The replication sets, if any, the user defined in filter `replication_sets` under `Table Filter`.

If the intersection among all 3 lists of replication sets is empty, which is the case for the BDR Witness, then LiveCompare will log this:

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

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

Differences in a BDR cluster

LiveCompare will make changes to the local node only; it is important that corrective changes do not get replicated to other nodes.

When `logical_replication_mode = bdr`, LiveCompare will initially check if a replication origin called `bdr_local_only_origin` already exists (the name of the replication origin can be configured by adjusting the setting `difference_fix_replication_origin`). If a replication origin called `bdr_local_only_origin` does not exist yet, then LiveCompare creates it on all BDR connections.

IMPORTANT: Please note that BDR 3.6.18 introduced the new pre-created `bdr_local_only_origin` replication origin to be used for applying local-only transactions. So if LiveCompare is connected to BDR 3.6.18, it won't create this replication origin.

LiveCompare will generate apply scripts considering the following:

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

After applying LiveCompare fix script to a BDR node, it will be possible to get exactly which rows 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);
```

(Note that deleted rows are no longer visible.)

Please note that LiveCompare requires at least a PostgreSQL user with `bdr_superuser` privileges in order to properly fetch metadata.

All steps above involving replication origins only applied to output script, if the PostgreSQL user has `bdr_superuser` or PostgreSQL superuser privileges. Otherwise, LiveCompare will generate fixes without associating any replication origin (transaction replication is still disabled using `SET LOCAL bdr.xact_replication = off`). However, it is recommended to use a replication origin when applying the DML scripts, because otherwise LiveCompare will have the same precedence as a regular user application regarding conflict resolution. Also, as there will not be any replication origin associated to the fix, the query above to list all rows fixed by LiveCompare can not be used.

Between BDR 3.6.18 and BDR 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 does not 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`), so the fix transaction applied by LiveCompare always loses a conflict, if any.

The functions above require a `bdr_superuser` rather than a PostgreSQL superuser. Starting from BDR 3.7.0, those functions are deprecated. LiveCompare then will, if running as a PostgreSQL superuser, use the following functions instead, to perform the same actions as above:

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

If a PostgreSQL superuser is not being used, then LiveCompare will include only the following in the generated DML transaction:

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

Conflicts in BDR

LiveCompare has an execution mode called `conflicts`. This execution mode is specific for BDR clusters. It will only work in BDR 3.6 or BDR 3.7 clusters.

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

Having said that, `conflicts` execution mode is expected to run much faster than `compare` mode, because it will just inspect specific tuples from specific tables. At the same time, it's not as complete as `compare` mode, because of the same reason.

The main objective of this execution mode is to check that the automatic conflict resolution which is being done by BDR is consistent among nodes, i.e., after BDR resolving 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 checking and ensuring consistency, with a good balance between time vs result.

Conflict example

Imagine on `node3` we 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;
```

We 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
```

Which 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`, this is what we see:

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

The BDR cluster is divergent.

Now in order to detect and fix such divergence, we could execute LiveCompare in `compare` mode, but depending on the size of the comparison set (imagine table `tbl` is very large), that can take a long time, even hours.

This is exactly the situation where `conflicts` mode can be helpful. In this case, the `delete_missing` conflict is visible only from `node3`, but LiveCompare is able to 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 will then check the current row version in all nodes in the cluster.

So we create a `check.ini` file to set `all_bdr_nodes = on`, i.e., 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:

```
2ndq-livecompare check.ini --conflicts
```

After the execution, in the console output, you will 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/` (being `X` the number of the current comparison session), LiveCompare will write the file `conflicts_DAY.out` (replacing `DAY` in the name of the file with the current day), showing the main information about all divergent conflicts.

If you connect to database `liveoutput`, you will be able to 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;
```

You will see 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 has compared the conflict and found the nodes to be currently divergent in the tables and rows reported by the conflict. View `livecompare.vw_conflicts` shows information about all conflicts, including the non-divergent ones.

LiveCompare will also automatically generate DML script `./lc_session_X/apply_on_the_node3_DAY.sql` (replacing `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 on the other 2 nodes the row does not exist. By default, LiveCompare suggest the DML to fix based on the majority of the nodes.

If you run this DML script against `node3`:

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

You will get the BDR cluster consistent again.

As the `--conflicts` mode comparison is much faster than a full `--compare`, it is highly recommended to schedule a `--conflicts` comparison session more often, to ensure conflict resolution is providing cluster-wide consistency.

Please note that, in order to be able to see the data in `bdr.conflict_history` in BDR 3.7 or `bdr.apply_log` in BDR 3.6, you should run LiveCompare with an user that is `bdr_superuser` or is a PostgreSQL superuser.

Mixing technologies

Please note that metadata for `node_name` and `replication_sets` are fetched in the `Initial Connection`. So it should be a pglogical- and/or BDR-enabled database.

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

It is possible to perform mixed technology comparisons, for example:

- BDR 1 node versus BDR 3 node;
- BDR 3 node versus vanilla Postgres instance;
- Vanilla Postgres instance versus pglogical node.

8 Oracle Support

LiveCompare can be used to compare data from an Oracle database against any number of PostgreSQL or BDR databases.

For example, you can define `technology = oracle` in a data connection. Other settings can then be used to define the connection to Oracle:

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

All other data connections are required to be PostgreSQL.

Here is a simple example of comparison between an Oracle database versus a PostgreSQL database:

```
 [General Settings]
logical_replication_mode = off
full_comparison_mode = on
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`

It is possible to disable schema-qualified table names by setting `schema_qualified_table_names = off`. This can be done only if `oracle_user_tables_only = on`, which means that LiveCompare will search only on tables that belong to the Oracle user that is 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
full_comparison_mode = on
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`, then you can use non-qualified table names in the `Table Filter`, `Row Filter` and `Column Filter` too.

Also please note that the `Output Connection` is required to write progress and reporting information from LiveCompare.

If you need to compare a BDR database against Oracle, and you want to take advantage about `Initial Connection`, `node_name` and `replication_sets` features (as explained earlier), then you can point the last data connection to Oracle, like this:

```
[General Settings]
logical_replication_mode =
bdr
full_comparison_mode = on
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'
```

It is also possible to compare a whole BDR cluster against a single Oracle database, for example:

```
[General Settings]
logical_replication_mode =
bdr
full_comparison_mode = on
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

Please note that LiveCompare works on PostgreSQL databases out-of-the-box, without needing to install any additional software.

But in order to be able to connect to Oracle, LiveCompare 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

The Python module cx_Oracle is required to be installed and available on your system so that LiveCompare is able to connect to an Oracle Database.

Currently cx_Oracle is not installable from Linux distribution repositories, so please follow the instructions in cx_Oracle website to install it on your system.

As LiveCompare is recommended to be executed under the `postgres` operating system user, then it is possible to install the `cx_Oracle` module through PIP only for the `postgres` user, with the following command:

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

Differences

If LiveCompare finds any difference, it will generate a DML script only to be applied on the PostgreSQL connections. A DML script to be applied in the Oracle connection is not generated.

BLOB and CLOB Data Types

LiveCompare is able to 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 should be of type `bytea`.

However, by default LiveCompare does not handle BLOB/CLOB fields if they are in the primary key, or if the table has no primary key. If that's the case, then the table will be ignored, and in LiveCompare logs you will see a message like this:

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

It is possible to workaround this behaviour by telling LiveCompare to ignore BLOB/CLOB fields if table has no primary key, by enabling these 2 settings in the `General Settings` section:

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

Incompatible Collation

On Oracle, generally we have the following initialization parameters 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, which can lead to false positives.

To workaround that, it is possible to 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 the user has set it to any other value.

Common Hash

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

- The row is fully represented as text, by concatenating all column values;
- On 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 will automatically fall back to `row_hash` and then `full_row`, as it would on a Postgres versus Postgres comparison.
- The BLOB, CLOB and NCLOB fields on Oracle are limited to the first 2000 characters only (`comparison_algorithm = full_row` allows comparison of the entire BLOB and CLOB);

- On Oracle, the full row representation should 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 appears in the logs:

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

Further LiveCompare versions will fall back to `full_row` comparison on these specific tables. For now, a workaround would be to configure a separate comparison sessions on these tables only, using `comparison_algorithm = full_row`.

Also note that the Common Hash requires the `standard_hash` function on Oracle, which is available only on Oracle 12c and newer. On Oracle 10g and 11c, please use `comparison_algorithm = full_row`.

9 Settings

General Settings

- `logical_replication_mode`: Affects how the program interprets connections and table filter settings (see more details below), and also what requirements to check for in the connections before starting the comparison. Currently the possible values are:

  ```
   - `off`: Assumes there is no logical replication between the databases;

  - `native`: Assumes there is native logical replication between the
  databases. Enables the usage of the `Table Filter -> publications`
  setting to specify the list of tables to be used. Requires PostgreSQL 10+ on
  all databases.

  - `pglogical`: Assumes there is pglogical replication between the databases.
  Enables the usage of the `Table Filter -> replication_sets` setting to
  specify the list of tables to be used. Also enables the usage of `node_name`
  to specify the data connections, which require setting the `Initial
  Connection` that is 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 BDR cluster.
  Enables usage of `Table Filter -> replication_sets` setting to specify list
  of tables to be used. Also enables usage of `node_name` to
  specify the data connections, which require setting `Initial Connection`
  that is 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 it is possible to specify only the Initial Connection (see below) and let LiveCompare build the connection list based on the current list of active BDR nodes. Default: `off`.

- `max_parallel_workers`: Number of parallel processes to be considered. Each process will work on a table from the queue. Default: `2`.

Important: Each process will keep N+1 open connections: 1 to each data connection and another 1 to the output database.

- `buffer_size`: Number of rows to be retrieved 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 3 modes, and by 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 a 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 may hang for a few seconds waiting for the data to be materialized), so the whole table data consumes RAM and can be stored on Postgres side disk as temporary files. All that impact can be decreased by using `parallel_chunk_rows` (set to 10000000 by default), and speed can be improved 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_stataments` , 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 potentially can cause bloat, `VACUUM` and replication issues on Postgres, but such impact is higher because `WITHOUT HOLD` cursors require an open transaction for the whole comparison session (this will be lifted in further versions). As the snapshot is held for the whole comparison session, comparison results might be helpful depending on your use case. As the query is not materialized, memory usage and temp file generation remains low. Asynchronous data fetch is not allowed. In terms of performance, this mode is slower for the general use case, but for large tables it can be the faster. It's recommended 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 may not happen when using server-side cursors on a busy database. So it's recommended to use `prepared_statements` for production, high load servers; and either `server_side_cursors_*` settings for testing, migration scenarios, and low load servers. The best strategy would probably mix `server_side_cursors_without_hold` for very large tables, and `prepared_statements` for the remaining tables. Refer to the table below for a comparison on 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 will always be fetch from Oracle using direct queries with `LIMIT` , without using prepared statements or cursors.

- `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 will have no more than `parallel_chunk_rows` rows. Setting it to any value < 1 disables table splitting. If any connections are not PostgreSQL, then table splitting is disabled automatically by LiveCompare. Default: 10000000.

- `parallel_data_fetch` : If data fetch should be performed in parallel (i.e., using async connections to the databases). Improves performance of multi-way comparison. If any data connections are not PostgreSQL, then this setting is automatically disabled. It's only allowed 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.

- `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 for a specific row the
hash does not match, then for that specific row, LiveCompare will fallback
to `full_row` algorithm (i.e., compare row by row). If any data connections
is not 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", i.e., 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 2-level Merkle Tree. If the block hash
matches, then LiveCompare advances the whole block (this is why this
comparison algorithm is faster than `row_hash`). If block hash does not
match, then LiveCompare falls back to `row_hash` and performs comparison row
by row in the buffer to find the divergent rows. This is the default value.
```

- `min_time_between_round_saves` : Time in seconds to wait before updating each round state when the comparison algorithm is in progress. Note that when the round finishes, LiveCompare always updates the round state for that table. Default: 60 seconds.

Important: If the user cancels execution of LiveCompare by hitting `Ctrl-c` and starts it again, then LiveCompare will resume the round for that table, starting from the point where the round state was saved.

- `stop_after_time` : Time in seconds after which LiveCompare will automatically stop itself as if the user had hit `Ctrl-c` . The comparison session that was interrupted, if not finished yet, can be resumed again by passing the session ID as argument in the command line. Default is `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 is `simple_majority` .

- `difference_required_quorum` : If `consensus_mode = quorum_based` , then this setting specified the minimum quorum is required to decide which connections are divergent. Should be a number between 0.0 and 1.0 (0.0 means no connection is required while 1.0 means all connections are required, both cases are extreme and should not be used). 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`) that should be considered as source of truth. It is only used 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` should be defined in the .ini file or `all_bdr_nodes = on` and only the `Initial Connection` is defined, while `node1` and `node2` should be valid BDR node names.

- `difference_tie_breakers` : Comma-separated list of connection names (or node names, if `logical_replication_mode = bdr` and `all_bdr_nodes = on`) that should 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` should be defined in the .ini file or `all_bdr_nodes = on` and only the `Initial Connection` is defined, while `node1` and `node2` should be valid BDR node names. Default is to not consider any connection as tie breaker.

- `difference_statements` : Controls what kind of DML statements will be generated by LiveCompare. 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` will be allowed in the difference report. Default: `on` .

- `difference_statement_order` : Controls order of DML statements that will be generated by LiveCompare. 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 BDR databases, for difference LiveCompare will create a specific replication origin if it doesn't exist yet, then use the replication origin to create apply script with DML fixes. The setting `difference_fix_replication_origin` specifies the name of the replication origin used by LiveCompare. If the user doesn't set any value for this setting, then LiveCompare will automatically set `difference_fix_replication_origin = bdr_local_only_origin` . Note that the replication origin that LiveCompare creates is not dropped to allow verification after the comparison, but if needed the replication origin can be manually dropped later. Requires `logical_replication_mode = bdr` .

IMPORTANT: Please note that BDR 3.6.18 introduced the new pre-created `bdr_local_only_origin` replication origin to be used for applying local-only transactions. So if LiveCompare is connected to BDR 3.6.18, it won't create this replication origin, and it is recommended that the user should not try to drop this replication origin.

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

```
 - If the divergent connection is BDR 3.6.7, add
`SET LOCAL bdr.xact_replication = off;`
- Add commands that setup transaction to use the replication origin
specified in `difference_fix_replication_origin`.
```

- `show_progress_bars` : Determines whether or not progress bars should be shown 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 will be created. Default: `livecompare` .

- `hash_column_name` : Every data fetch will contain a specific column which is 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. Default: `off` . To be enabled, requires `logical_replication_mode` set to `pglogical` or `bdr` .

- `column_intersection` : When this setting is enabled, for a given table that is being compared, LiveCompare will only work 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 will check if columns are equivalent on the table on all connections, and abort the comparison of the table if there are any column mismatches. Default: `off` .

Important: If table has PK, then the PK columns are not allowed to be different, even if `column_intersection = on`.

- `oracle_ignore_unsortable`: When enabled, tells LiveCompare to ignore columns with Oracle unsortable data types (BLOB, CLOB, NCLOB, BFILE) if column is not part of the table PK. If enabling this setting, it is recommended to also enable `column_intersection`.

- `oracle_user_tables_only`: When enabled, tells LiveCompare to fetch table metadata only from the Oracle logged in user, which 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 comparison of tables without using schema-qualified table names: on Oracle x Postgres comparisons, it requires also enabling `oracle_user_tables_only`, while 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 versus Postgres databases (in this case users should set `force_collate = C`). Will assume value `C` if comparing mixed technologies (like Oracle vs PostgreSQL) and no collation is specified. Default: `off`.

- `work_directory`: path to the `LiveCompare` working directory. The session folder containing output files will be created in such directory. Default: `.` (current directory).

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

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

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

- `session_replication_role_replica`: when enabled LiveCompare will use `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 super user, otherwise will take no effect. Defaults to `off`.

- `split_updates`: when enabled LiveCompare will split `UPDATE` divergences, i.e., instead of generating a `UPDATE` DML, it will generate corresponding `DELETE` and `INSERT` in the apply script. Defaults to `off`.

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

Initial Connection

The initial connection is used only when `logical_replication_mode` is set to `pglogical` or `bdr`, and is used only when the program starts, to fetch DSN from node names, if the user has set data connections using only the `node_name` setting.

- `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 the user set `dsn` instead of all other connection information. Default: `livecompare_initial` .

Output Connection

The output connection specifies where LiveCompare will create 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 the user set `dsn` instead of all other connection information. Default: `livecompare_output` .

Data Connection

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

Similarly to the `Initial Connection` and `Output Connection` , a "data connection" is defined in a named section. The section name should be of the form `Name Connection` , being `Name` any single-worded string starting with an alphabetic character. In this case, whatever the user fills in `Name` is called the "Connection ID" of the data connection. It is also required that each data connection has an unique Connection ID in the whole list of data connections.

If `logical_replication_mode = bdr` and `all_bdr_nodes = on` , then the user is not required to specify any data connection, because LiveCompare will build the data connection list by fetching BDR 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 recommend not to use this, but in some legacy connections it might be required.
- `application_name` . Application name. Can be used even if the user set `dsn` or `node_name` instead of all other connection information. Default: `livecompare_<Connection ID>` .
- `start_query` : Arbitrary query that is executed each time a connection to a database is open.
- `fetch_fk_metadata` : If LiveCompare should gather metadata about foreign keys on the connection. Default: `on` .

Table Filter

If omitted or left empty, this section from the `.ini` file will mean that LiveCompare should be executed against **all** tables in the **first** database.

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

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

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

Requires `logical_replication_mode = native`.

- `replication_sets`: When using pglogical or BDR, you can filter specific replication sets, and LiveCompare will work only on the tables associated to those replication sets. The variable `set_name` can be used 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 will work only on the tables that belong to those schemas. The variable `schema_name` can be used 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'
```

Please note that, in any conditional expression, the `%` character should be escaped as `%%`.

The table name should be schema-qualified, unless `schema_qualified_table_names` is disabled. For example, it's possible to filter only a specific list of tables:

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

If you have disabled general setting `schema_qualified_table_names`, then you should 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: Please note that if two or more schemas that were set on `search_path` contains a table if the same name, just the first one found will be considered in the comparison.

The `Table Filter` section can have a mix of `publications`, `replication_sets`, `schemas` and `tables` filters, and LiveCompare will consider 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'
```

Also please note that 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 does not exist in any other database, then you will see something like this in the logs, and the comparison for that specific table will be 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 does not exist in the first database, then it won't be considered in the comparison, even if you didn't apply any table filter.

A comparison for a specific table will also be skipped if the table column names are not exactly the same (unless `column_intersection` is enabled), and in the same order. An appropriate message will be included in the log file as well.

Currently LiveCompare does not check if data types nor constraints are the same on both tables.

IMPORTANT: please note that `conflicts` mode doesn't make use of table filter.

Row Filter

In this section you can apply a row-level filter to any table, so LiveCompare will work 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 should 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 will work only in the rows that satisfy the clause `id = 10`, while for the table `public.table2`, only rows that satisfy `logdate >= '2000-01-01` will be considered in the comparison.

If you have disabled general setting `schema_qualified_table_names`, then you should 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 would 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, it's possible to 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 will try to apply the filter but will fail. So the comparison for this specific table will be skipped, and an exception will be 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 will be silently ignored.

IMPORTANT: please note that `conflicts` mode doesn't make use of row filter.

Column Filter

In this section you can apply a column-level filter to any table, so LiveCompare will work only on the columns that are not part of the column filter.

You can write a list of tables under this section, one table per line (all table names should be schema qualified unless `schema_qualified_table_names` is disabled). For example, considering 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 will work only in the columns `column2`, `column4` and `column5`, filtering out `column1` and `column3`, while for the table `public.table2`, only the columns `column2`, `column3` and `column4` will be considered in the comparison, filtering out `column1` and `column5`.

If you have disabled general setting `schema_qualified_table_names`, then you should 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, column doesn't exist in the given table, then LiveCompare will log a message about the columns that could not be found and ignore them, using 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 will be silently ignored.

IMPORTANT: Please note that if a column specified in a `Column Filter` is part of the table PK, then it won't be ignored in the comparison. LiveCompare will log that and ignore the filter of such column.

IMPORTANT: please note that `conflicts` mode doesn't make use of column filter.

Conflicts Filter

In this section you can specify a filter to be used in `--conflicts` mode while fetching conflicts from BDR nodes. You can build any SQL conditional expression, and use below 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 the conflict
- `conflict_resolution`: the resolution which was applied
- `nspname`: schema name of the involved relation
- `relname`: relation name of the involved relation

You must use `conflicts` attribute under the section. Please find an example below:

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

By adding above piece of configuration to your INI file, LiveCompare would fetch just conflicts that are of type `update_missing`, and related to tables under schema `my_schema` while querying for conflicts in each of the BDR nodes.

IMPORTANT: Please note that this section is exclusive 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 Licence (MPL) v. 2.0 - Exhibit A

MIT License (MIT)

cx_Oracle

Apache License

```
                                 Apache License
                               Version 2.0, January 2004
```

(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.

Tabulate

OmniDB