



EDB

LiveCompare

Release 2.2.0

EDB

Jun 16, 2022

CONTENTS

1	Introduction	3
1.1	Comparison Performance	4
1.2	Security Considerations for the User	4
1.2.1	Release Notes	4
1.2.2	Requirements	25
1.2.3	Supported Technologies	27
1.2.4	Command-line Usage	28
1.2.5	Advanced Usage	32
1.2.6	BDR Support	35
1.2.7	Oracle Support	43
1.2.8	Settings	48
1.2.9	Licenses	59

© Copyright EnterpriseDB UK Limited 2019-2021 - All rights reserved.

INTRODUCTION

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.

1.1 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.

1.2 Security Considerations for the User

For PostgreSQL 13 and older, LiveCompare requires an user that is able to read all data being compared. PostgreSQL 14 introduced a new role `pg_read_all_data`, which can be used for LiveCompare.

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.

1.2.1 Release Notes

2.2.0 (2022-06-14)

New features

- Added `--dry-run` execution mode, which allows users to validate the `.ini` file and view some useful comparison information, without running comparison session (LIV-142, RT78462):
- Any abort messages received during the comparison session are printed in chronological order
- Displays the list of connections, including technology, version, if the connection is a tiebreaker or a source of truth, and if it's reachable;
- Displays the `Table Filter` if it is configured;

- Lists the tables that will be 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.

Improvements

- Support for SLES 12 on x86_64 (LIV-112).
- Support for SLES 15 on x86_64 (LIV-113).
- Updated the list of reserved words according to Postgres `kwlist` source code (LIV-141, RT80745).
- The main progress bar percentage is now using a float instead of an integer. Previously, the number was rounded up (LIV-138, RT80745).
- 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).
- Demoted hash mismatch log messages from `WARNING` to `DEBUG` (LIV-145).
- Logs will now include the `application_name` in the message (LIV-141).
- 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).
- Improved performance when generating the apply DML scripts when there is an increased number of divergences detected (LIV-159).

Bug fixes

- 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).
- 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).
- Fixed a corner case where unhandled exceptions could cause the comparison worker to hang (LIV-140, RT80745).
- Normalizing decimal values to avoid false positives when comparing Oracle versus Postgres (LIV-153).
- 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).
- Allow the same table to be configured in multiple filter sections (LIV-156).

2.1.0 (2022-03-31)

New features

- Support for Postgres-BDR 4 (LIV-131).
- 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).
- 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).

Other changes

- 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. More details are in the documentation for the setting (LIV-130).
- Demoted to DEBUG the log message about the number of processed rows from `CanAdvanceCursors` method (LIV-129).

Bug fixes

- Fixed an issue for Oracle versus Postgres comparisons of the `timestamp(6)` data type where failing with `ORA-01830` (LIV-127).

2.0.0 (2022-02-15)

Breaking changes

- Package has been renamed to `edb-livecompare` from `2ndq-livecompare` .
- Executable has been renamed to `livecompare` from `2ndq-livecompare` .

New features

- 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 (LIV-5).
- 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 which 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 will try 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).
- 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).
- 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).

Performance improvements

- 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).

Other improvements

- Added support to Oracle 21c (LIV-47).
- LiveCompare schema can now be added to a replication-enabled (BDR, pglogical or native logical replication) database (LIV-42).
- LiveCompare can use the new `pg_read_all_data` role in PostgreSQL 14 (LIV-73).
- Abort with a proper message if any database version is not supported.

Bug fixes

- Properly quote the unicode sequence `\u0000` to avoid an error when generating DML (LIV-98).
- Fixed an issue where the number of total rows was displayed incorrectly when the table was split into multiple round parts (LIV-14).
- Fixed an issue where empty BLOB on Oracle when compared against an empty bytea on Postgres was generating a false positive (LIV-103).
- Fixed an issue where connectivity issues were causing exceptions aborting the whole comparison session. Now LiveCompare is able to re-connect and continue the comparison if possible (LIV-84).
- Fixed an unhandled exception on the recheck mode if there are any divergences (LIV-107).
- 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).
- Fixed an issue where fields of `timestamp` data type were always generating a mismatching hash between Oracle and Postgres.
- Fixed an issue where ignored columns were still being considered in the common hash.

Deprecation notice

- Dropped support to Oracle 10g (LIV-47).

1.18.1 (2021-12-14)

Bug fix

- Fixed an issue where the internal round part column name setting was not being considered in asynchronous connections (LIV-95).

1.18.0 (2021-12-01)

Performance Improvements

- 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).
- 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 (LIV-85, LIV-86, RT75644, RT75675, RT75332).
- 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 (LIV-87).

Other Improvements

- Support for EDB Extended PostgreSQL 14 and EDB Advanced PostgreSQL 14 (LIV-71, LIV-72).
- Collecting the Oracle rowid which can be helpful to inspect divergences (LIV-90, RT75644, RT75675).
- Improved logging for the LiveRound comparison algorithm, which can be helpful to inspect divergences (LIV-91, RT75644, RT75675).

Bug fixes

- Fixed the issue where `oracle_fetch_fk_metadata` was not being honoured (LIV-92).
- 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).

1.17.0 (2021-10-01)

Improvements

- Added support to PostgreSQL 14 (LIV-66).
- Added support to Debian 11 (LIV-70).

Bug fixes

- Now `--recheck` will always require a session ID (LIV-76).
- Minor fixes to the documentation (LIV-26, LIV-68, LIV-80).

1.16.0 (2021-08-04)

Performance Improvements

- Now Oracle versus Postgres comparison sessions can use a common row hash based on MD5 that improves performance for the general use case. Previously, Oracle versus Postgres comparison was limited to `comparison_algorithm = full_row`, but now `comparison_algorithm = row_hash` or even `comparison_algorithm = block_hash` can be used. By default, it's set to `comparison_algorithm = block_hash`. Due to Oracle limitations, using the Common Hash, BLOB and CLOB fields are limited to 2000 characters, and the entire row can not be wider than 4000 characters. Tables not meeting the requirements for the Common Hash can be compared separately using `comparison_algorithm = full_row`. Please check the documentation for additional details (RT72038, LIV-61).
- Increased the default value for `buffer_size` from 256 rows to 4096 rows.
- Increased the default value for `parallel_chunk_rows` from 1 million rows to 10 million rows.

1.15.0 (2021-06-06)

Improvements

- Introduced conflicts filtering for `--conflicts` execution mode (LIV-40)
- LiveCompare will assume `force_collate = C` in mixed technologies comparison (like Oracle vs PostgreSQL) if no collation is specified (LIV-44).
- Besides support to EPAS 13, which was already added, added support to EPAS 11 and 12 (LIV-48).

Bug fixes

- LiveCompare is not getting stuck anymore in Oracle vs PostgreSQL comparisons (RT71243 / LIV-43)
- Fixed a bug where it was not possible to run `--recheck` on a comparison session that was created with `--conflicts` (LIV-50)

1.14.0 (2021-05-14)

New features

- New `--conflicts` execution mode, specific for BDR 3 clusters. For BDR 3.6, table `bdr.apply_log` is checked, and for BDR 3.7, table `bdr.conflict_history` is checked. 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. By checking only the BDR conflicts, the comparison is much faster than `--compare` mode, so it's feasible and recommended to be executed often to ensure cluster consistency (LIV-28).

Improvements

- LiveCompare is now able to ignore BDR nodes where a specific table does not exist, or the table does not receive any replicated data according to replication set configuration. Previously the comparison was performed on all connections, now the comparison consensus is able to automatically filter out nodes per table as needed. Consensus majority is still achieved among the nodes where the table exists, and where the table is associated with a replication set the node is subscribed to. This allows for comparison on BDR cluster that have different architectures defined by replication sets, including properly ignoring BDR Witness nodes (LIV-29).
- Improved message for `--help` showing a small description for all execution modes (LIV-28).
- Information about if the column is nullable or not is now also gathered in table metadata (LIV-28).
- Clarified in docs that unsortable data types are converted to `string` if the columns are part of the primary key (LIV-4).
- Clarified in docs that `oracle_ignore_unsortable` ignores unsortable columns which are not part of the PK in Oracle (LIV-26).
- Clarified in docs that `parallel_data_fetch` can be enabled (it is by default) when `data_fetch_mode` is set to `prepared_statements` (default) or `server_side_cursors_with_hold` (LIV-26).
- Clarified in docs that `max_parallel_workers` default value is 2 (LIV-26).

Bug fixes

- LiveCompare is now handling types as schema qualified names. Types out of `search_path` could not be found by LiveCompare before this fix (LIV-35).

Deprecation

- Removed usage of `bdr.difference_fix*` functions that are deprecated on BDR 3.7. Those functions are still included in the generated DML scripts for BDR 3.6, as they require a `bdr_superuser` rather than a PostgreSQL superuser. This is clarified in documentation (LIV-1,LIV-31).

1.13.1 (2021-04-14)

Bug fixes

- Fixed issues in support for Oracle data types RAW and ROWID (RT69656).
- Fixed an unhandled exception when a binary object field from Oracle was NULL (RT69656).
- Fixed an issue on Oracle x Postgres comparison support involving dollar-quoted strings (RT69656).
- Fixed a corner case where columns named as a particular PostgreSQL reserved key word were being quoted and not being considered in the comparison (RT69656).

1.13.0 (2021-02-25)

Performance improvements

- Implemented new setting `data_fetch_mode` , which applies only for PostgreSQL connections. It can assume one of the following values: - `prepared_statements` : Default behavior in this and previous versions, using prepared statements to fetch data. Smaller footprint on the database, good performance for the general use case. It's strongly recommended for production databases. - `server_side_cursors_with_hold` : Uses a server-side cursor WITH HOLD to fetch data, which can hold back `xmin` and increase memory and temp file usage on the database, as each chunk of data is materialized in temp files before being sent to LiveCompare. As a result it can be faster for multiple small tables, and is recommended for testing and migration scenarios. - `server_side_cursors_without_hold` : Uses a server-side cursor WITHOUT HOLD to fetch data, which can hold back `xmin` , but memory and temp file usage on the database is lower than WITH HOLD . It can be faster for large tables and is recommended for low load databases.

Other improvements

- Implemented new setting `abort_on_setup_error` , when enabled tells LiveCompare to abort the whole comparison session if there is any error when setting up a comparison round for any table. Disabled by default, which means that LiveCompare will log and ignore any tables where the setup phase failed for any reason.
- Because of new setting `abort_on_setup_error` , settings `setup_max_attempts` and `setup_min_interval_between_attempts` are now obsolete.

1.12.0 (2021-02-11)

New Features

- Implemented new setting `session_replication_role` . If enabled will set `session_replication_role` to `replica` in beginning of apply scripts (RM20247, RT69742).
- Implemented new setting `split_updates` . If enabled will split UPDATE statements into corresponding DELETE and INSERT statements (RM20355, RT69742).
- New connection setting `fetch_fk_metadata` , which tells if metadata about foreign keys should be fetched on the specific connection. Enabled by default.
- New general setting `oracle_fetch_fk_metadata` , which overrides the value of `fetch_fk_metadata` for the Oracle connection. Disabled by default (RM20244, RT69656).
- New general setting `float_point_round` , which when tells LiveCompare to round by the specified decimal digits when comparing float point values. Disabled by default (RT69381).

Removed features

- Removed `--fix` execution mode. User are encouraged to review the DML script LiveCompare generates.

Improvements

- Apply scripts generated by LiveCompare now contain also `difference_fix_start_query` at the beginning and sets tables owner before applying DML.
- Now `comparison_algorithm = full_row` honors `parallel_chunk_rows` and splits the table into multiple workers if technology being compared in all connections is PostgreSQL. To disable table splitting, set `parallel_chunk_rows = 0` .
- Filtering out tables in system schema `sys` when comparing EPAS (RM20392).
- Increased default value of `max_parallel_workers` from 2 to 4.

Bug fixes

- Fixed a performance degradation when `comparison_algorithm = block_hash` and there were divergences in the data chunk a parallel worker was processing.
- Fixed an issue where the default randomly generated string for `custom_dollar_quoting_delimiter` was causing LiveCompare to crash on FIPS-enabled operating systems (RT70032, RM20378).
- Fixed an issue where `oracle_ignore_unsortable` or `column_intersection` could remove all columns, and LiveCompare would still try to perform a comparison. Now LiveCompare aborts comparison on those tables and logs a proper message (RT69656).
- Fixed a corner case where LiveCompare was not using the `custom_dollar_quoting_delimiter` to represent PK values (RT69656).
- Fixed a bug where running LiveCompare on `--recheck` mode could hurt a constraint on its output schema due to difference ordering.

1.11.0 (2021-01-19)

Improvements

- New setting `difference_statement_order` for the order of the DML commands generated by LiveCompare, by default set to `delete_update_insert` (RT69742, RM20245).
- Support for EnterpriseDB PostgreSQL Advanced Server - EPAS 13.
- Packaging for PostgreSQL 13 on CentOS and RHEL systems.

Bug fixes

- Fixed a bug where if a cursor was not initialized yet because its minimum PK value was higher than the maximum PK value of a cursor that has already reached end of data, the former cursor would be initialized but its data would not be read until the end (RT69742, RM20249).
- Fixed a bug where LiveCompare was not allowing `Initial Connection` and `node_name` when `logical_replication_mode = pglogical` (RM19795).
- Fixed a bug where LiveCompare was forcing the need of `oracle_user_tables_only = on` when `schema_qualified_table_names = on` instead of forcing it when `schema_qualified_table_names = off` (RT69656).
- Fixed a bug where strings were not properly quoted if `comparison_algorithm = full_row` (RM20251).
- Fixed a bug where strings were not properly quoted in the generated DML.

1.10.1 (2020-10-29)

Bug fixes

- Improved performance for fetching minimum and maximum primary key values for each table, which is done in the initial metadata fetch phase (RM19681, RT69088).
- Now comparison sessions can be resumed and execution modes `--fix` and `--recheck` can be executed against previously created comparison sessions in a different LiveCompare version, if it is on the same major version. For example: if a session was created on 1.10.0, customer upgrades LiveCompare to 1.10.1, and then runs `--fix`, LiveCompare will allow the `--fix` mode to execute, because the session was created in the same major version (RM19686, RT69088).
- Fixed a single quote issue when saving `Row Filter` in LiveCompare metadata (RM19682, RT69088).
- LiveCompare default dollar quoting delimiter (when `custom_dollar_quoting_delimiter` is not set) is now a fixed hash string to tackle an issue where resuming a comparison session was not working because the random setting value changed between executions (RM19684, RT69091).
- Setting `custom_dollar_quoting_delimiter` can now be explicitly disabled with `off`, which means LiveCompare will use the default hash string as a delimiter. (RM19685, RT69091).

1.10.0 (2020-10-02)

Improvements

- New general setting `custom_dollar_quoting_delimiter`, to tell LiveCompare how strings should be quoted when generating Postgres DML commands. When not set (default), LiveCompare generates a random string to use as dollar quoting delimiter (RT68950, RM19159).
- LiveCompare now checks that neither settings nor connections attributes were changed between the first execution of a session (session creation) and following ones (`resume`, `--recheck` or `--fix`) (RM17639).
- LiveCompare now can be softly stopped when running in background or in a different shell window (RM18564).

Bug fixes

- LiveCompare was not updating number of differences correctly while running in `--recheck` mode after user had applied fixes manually using the `apply_on_the_<node>.sql` file (RM19158).
- If LiveCompare was executed multiple times in `--recheck` mode, it would show replicated entries in `apply_on_the_<node>.sql` files and also in the summaries a replicated count of differences/fixes (RM19176).
- A large number of divergences would cause an `integer out of range` error in summary views (RM18705).

1.9.1 (2020-08-13)

Bug fixes

- We are now handling POSIX constants that may not exist on a given platform (RM18197).
- When dealing with partitioned tables, if the user set a table filter to remove any of the partitions, LiveCompare would compare both master table and all other children except for the filtered. That is now fixed. (RM16912).

1.9.0 (2020-08-06)

New features

- Setting `full_comparison_mode = on|off` was deprecated and replaced with a new setting `comparison_algorithm` which allows the new `block_hash` algorithm setting - this showed 50% performance gain in basic tests. Full list of supported algorithms are these: - `full_row` (same as old `full_comparison_mode = on`): Disables row comparison using hashes. It also disables table splitting, because it relies on a hash, so the setting `parallel_chunk_rows` is ignored and not tables are split. Full comparison, in this case, is performed by comparing the row column by column. If any data connections are not PostgreSQL, then LiveCompare automatically sets `comparison_algorithm = full_row`. - `row_hash` (same as old `full_comparison_mode = off`): 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). This setting is allowed only if all data connections are PostgreSQL. - `block_hash` (new implemented comparison algorithm): 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 setting is allowed only if all data connections are PostgreSQL. This is the default value (RM14146).

- LiveCompare now is able to do a few attempts on each table setup before exiting with an error. Two new settings were added to configure this behavior: `setup_max_attempts` (defaults to 3) and `setup_min_interval_between_attempts` (defaults to 30 seconds) (RM17518).

Improvements

- LiveCompare output is now printed in the output file while using `>` to redirect output and also displayed while executing LiveCompare through `ssh` (RM17372).
- Consensus was re-factored for improved logging and debugging (RM17107).
- At the end of the comparison session, LiveCompare now shows counter for issues that were found, depending on the log level (RM18196).
- DML is now being generated for all nodes in case of a Tie (RM16656).
- LiveCompare is now showing the maximum number of ignored columns on the table summary, if any divergences are found on a given table (RM16581).
- LiveCompare is now aborting with proper message in case `Output` or `Initial` (if specified). connections are not reachable. Also aborting with proper message if less than 2 data connections were reachable. If 2 or more are reachable, will compare just that ones (RM14030).
- Debian and Ubuntu packages now require `python3-setuptools` (RM17232).
- Clarified in docs about minimum Python version and Linux distributions supported (RM17232).
- When dealing with partitioned tables, LiveCompare will prefer scanning each partition instead of using the master table. This way we achieve a better estimation of row count and also better split of job between workers (RM16912).
- Using POSIX standard return codes instead of difference count (RM18197).
- Added a sample `config.yml` example in docs showing how to install LiveCompare using `TPAexec` (RM15312).
- Improved logging for `--recheck` and `--fix` modes (RM15768).
- If any exception happens, Python stack trace is included in logs (RM17158).
- Included in the docs some considerations about LiveCompare connecting to PostgreSQL through `PgBouncer` (RM18019).
- Added `round_id` and `round_part` fields to the `vw_running_processes` view, which helps checking comparison progress when LiveCompare execution is scheduled as a cron job (RM16910).

Bug fixes

- Fixed a racing condition in Consensus that caused a tuple concurrently updated in the `Output Connection` (RM17107).
- Fixed an issue where `num_processed_rows` was reported higher than the real value (RM17373).
- LiveCompare is not reporting successfully executed anymore if any kind of problems were found during execution (RM17289).
- Enforced having at least one worker in order to avoid hanging if the user set `max_parallel_workers <= 0` (RM17374).

- Fixed number format for large integers on printed tables. LiveCompare is now displaying the entire number instead of using scientific notation (RM16683).
- Fixed an issue where `--recheck` mode was reporting incorrect difference statuses after `--fix` mode was executed (RM15953).
- Fixed an issue where the list of ignored columns was being incorrectly reported in table metadata (RM17663).
- Fixed an issue where re-running LiveCompare with a different source of truth or tie breakers would cause divergences being incorrectly reported as ties (RM17909).

1.8.0 (2020-06-15)

New features

- New setting `difference_sources_of_truth`, used to tell which connections should always win consensus. Requires that `consensus_mode` is set to `source_of_truth` (RM15952).
- New setting `work_directory`, which indicates where the session folder will be created. Useful to run LiveCompare scheduled as a cron job (RM16476).

Improvements

- Order of DML commands written to the difference fix DML script now takes foreign keys into account. Same order is also used by the `--fix` execution mode (RM15766).
- LiveCompare now checks `difference_tie_breakers` and `difference_sources_of_truth` against the list of known connection IDs. If an unknown connection ID is specified, an error will be shown to the user and the comparison will be aborted (RM16622).
- Added support for CentOS 8 and Ubuntu 20.04 (RM14480, RM14619, RM14974).
- When printing regular tables, it would print an empty cell when data is null. Now prints `[null]` instead (RM16686).

Bug fixes

- Fix minor issues detected by Coverity scanner (RM16623).
- Now deals correctly when the main script is invoked with only three parameters, making the correct distinction whether it is a session ID or an execution mode.
- When printing tabular data in transposed format, if data was null then the whole field would be omitted. This issue is now fixed (RM16686).

1.7.0 (2020-05-18)

Breaking changes

- Implemented LiveCompare execution modes: `--compare` : default execution mode. Only performs comparison, there is no difference re-check/fix thread executing in parallel anymore. During comparison, each difference found is stored in the `difference_log` table for later re-check and optional automatic fix. `--recheck` : Can be executed against a session that was already created by the `compare` mode any number of times. Re-checks differences one by one and updates the `difference_log` table. `--fix` : Can be executed against a session that was already created by the `compare` mode. Re-checks differences one by one and tries to automatically fix them, updating the `difference_log` table.
- In compare mode, the view that holds the list of divergences is `vw_open_differences` , while for re-check and fix modes, the view is `vw_differences` .
- In automatic fix mode, for BDR $\geq$ 3.6.18, LiveCompare uses the new pre-created replication origin called `bdr_local_only_origin` (RM14699).
- Removed settings `difference_mode` , `min_time_between_difference_checks` , `max_difference_check_attempts` and `difference_check_nap_time` .
- Removed settings `live_mode` , `min_time_between_rounds` and `max_tail_rounds_before_full_round` .
- If setting `show_progressBars` is enabled (it is by default) and Python module `tqdm` is $<$ 4.16.0, LiveCompare aborts explaining how to upgrade.

New features

- Implemented new general setting `column_intersection` , disabled by default. When this setting is enabled, LiveCompare allows comparison of tables containing different set of columns, as long as PK columns are the same. The set of columns considered in the comparison is the intersection of columns existing on the table on all connections (RM14147, RT67064).
- Implemented new section `Column Filter` , where for each table it is possible to define a comma-separated list of columns that should be ignored in the comparison. Columns that are part of the PK can't be ignored. The format of this section is one table per line, similarly to the `Row Filter` section (RM14629, RT67064).
- Implemented new general setting `oracle_ignore_unsortable` , disabled by default. When enabled, tells LiveCompare to ignore columns with Oracle unsortable data types (BLOB, CLOB, NCLOB, BFILE) if table has no PK. If enabling this setting, it is recommended to also enable `column_intersection` (RT67064).
- Implemented new general setting `oracle_user_tables_only` , disabled by default. When enabled, tells LiveCompare to fetch table metadata only from the Oracle logged in user, which is faster. Also, `Table Filter -> tables` can be filtered by table name without schema name (RT67064).
- Implemented new general setting `schema_qualified_table_names` , enabled by default. 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. (RM14901, RT67042).
- When `schema_qualified_table_names` is enabled and `start_query` is not set (default), then LiveCompare uses `start_query` to clear `search_path` in order to protect from CVE-2018-1058 (RM15391).
- Implemented new general setting `force_collate` , by default set to `off` , which means that a collation will not be forced in PostgreSQL. When set to a valid collation name, it is useful to compare Postgres databases that have

different collation or Oracle versus Postgres databases if Postgres has a collation other than C (in this case users should set `force_collate = C`) (RM15016, RT67064).

- Implemented new general setting `fetch_row_origin`, disabled by default. When this setting is enabled, LiveCompare fetches the BDR/pglogical origin name for each divergent row (RM14487).
- If an exception is found during a comparison, now LiveCompare aborts the comparison round for the specific table writing the error in new column `rounds.round_error`, putting the failed worker back into the pool (RT67064).
- LiveCompare general progress bar now shows the number of tables aborted due to errors during the comparison.

Improvements

- Opening connections at the beginning and re-using database connections (RT67042).
- Clarified error message when user does not have permissions to read the configuration file (RT67064).
- Clarified in the docs that `Table Filter` and `Row Filter` require schema-qualified table names unless the general setting `schema_qualified_table_names` is disabled, and provided configuration examples (RM15311, RT67669).
- Aborting with error message if an unrecognized setting is found in the configuration file.
- Clarified in the docs about the number of connections required.
- Clarified in the docs about the dependency on the EPEL repository for CentOS/RHEL.
- Clarified in the docs on how to install the `cx_Oracle` Python module for the `postgres` operating system user.
- Added a `lc_` prefix to session directory and log file.
- If in any connection `technology = oracle` and Python module `cx_Oracle` is not found, LiveCompare aborts explaining how to install latest `cx_Oracle` for the current user.
- Automatically generated DML `*.sql` scripts now include a `SET LOCAL bdr.xact_replication = off;` clause for BDR.

Bug fixes

- Fixed an issue where an Oracle versus Postgres table comparison was being aborted due to a column name mismatch being incorrectly assessed because the column name is a reserved word in Postgres (RT67064).
- Oracle: Fixed an issue where a PK with a text column might generate an `ORA-00920: invalid relational operator` error (RT67064).
- Fixed an issue in Oracle and Postgres where a column of an unsortable data type was not being properly handled in `ORDER BY` clauses (RT67064).
- Fixed an issue in the DML generator where an `UPDATE` was not setting a column to `NULL`, because the setting `difference_allow_null_updates` was being misinterpreted.
- Fixed a corner case where the comparison cursor was not properly advancing.
- Fixed error message for when user tries to resume a comparison session that is already finished.

1.6.0 (2020-04-03)

- If PostgreSQL ≥ 11 , using built-in function `hashtextextended` instead of `hashtext` to split data among the comparison workers (RT67167, RM13664).
- Fixed an issue where the list of Oracle PK column names was having duplicate column names, resulting in an error and comparison being aborted (RT67064, RTM14145).
- If can't initially connect to data connections, now LiveCompare aborts the whole comparison session (RT67042).
- Fixed an issue where the round state was being saved too frequently and not honoring `min_time_between_round_saves`. Improves performance.
- Improved logging of connection and query issues.

1.5.0 (2020-03-13)

- Setting `parallel_data_fetch` is automatically disabled if one of the connections is Oracle, as Oracle does not support `parallel_data_fetch` (RM13714, RT67042 and RT67064).
- Increased maximum number of rounds that can be performed, rows that can be processed and differences that can be found in a single comparison session (data type from `integer` to `bigint`) (RM13664 and RT67167).
- Changed progress bars from ASCII to Unicode.
- Updating global progress bar time elapsed every 5 seconds.
- Removed rate and estimated time from global progress bar, and added number of connections.
- If Python module TQDM ≥ 4.16 is available, the global progress bar shows additional stats (number of differences found and automatic fixes applied).
- Fixed an issue where the maximum number of processes was limited by the number of tables scheduled to be compared, even if the table was split.
- Reduced log verbosity from INFO to DEBUG when getting table metadata.
- Including in log cases where a manual (Ctrl-C) or automatic interruption (`stop_after_time`) happens.

1.4.0 (2020-02-13)

- Implemented parallel data fetch to improve performance of multi-way comparison. Parallel data fetch is enabled by default but can be disabled by setting `parallel_data_fetch = off`.
- Fixed an issue where the row was local, i.e., its `xmin` was not associated to any replication origin (RT66906).

1.3.0 (2020-01-30)

- Multi-way comparison: LiveCompare is now able to compare any number of connections. Comparison is done by fetching data from all nodes at the same time. This allows determination of data inconsistencies based on consensus (both quorum-based or simple majority are supported) and an optional list of tie breaker connections. It is possible to see details on how LiveCompare worked using the new view `vw_consensus`.
- Added new general setting `all_bdr_nodes`, which when enabled allows the user to specify only the Initial Connection section that should point to any BDR node, and then LiveCompare will build the list of connections considering all active nodes in the BDR cluster. Please note that it requires that LiveCompare is able to connect to all BDR nodes using the node DSN as it can be seen in `bdr.node_summary` view.

- Added new general setting `consensus_mode` , which determines which connections (or BDR nodes) are considered correct when data comparison finds a divergence. Can be `simple_majority` or `quorum_based` . If `consensus_mode = quorum_based` , then the new setting `difference_required_quorum` (values between 0.0 and 1.0, default 0.5) is considered. Default is `consensus_mode = simple_majority` .
- Added new setting `difference_tie_breakers` , to help in cases where consensus can not determine correct connections or nodes in case of data divergence. Must be a comma-separated list of connection names, 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 `bdr_all_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.
- Multi-way comparison also allows connection names other than “Left” and “Right” in the connection section name. Backward compatibility is kept so users can still define `Left Connection` and `Right Connection` , but now only 2 connections require definition of `difference_tie_breakers` as explained above. Previously the “Left” connection was always considered as tie breaker, i.e., as correct when automatic difference fix was enabled.
- Multi-way comparison requires that only one of the connections is different than PostgreSQL, if any.
- Implemented new setting `stop_after_time` to allow LiveCompare to be manually interrupted after a number of seconds. By default `stop_after_time = 0` which means that LiveCompare will not automatically stop (only by manual Ctrl-c). LiveCompare can be manually stopped with Ctrl-c in all cases. Regardless of whether LiveCompare was manually or automatically interrupted, it can be resumed by passing the session ID as argument in the command line.
- New table `difference_fix` stores the exact DML LiveCompare executed (or tried to execute) on each data connection, the time and the error (if any). Scripts `applied_*.sql` now contain the same applied DML too.
- LiveCompare now stores table owner in table metadata.
- Changed difference fix transaction timestamp from 2000-01-01 to 2010-01-01. This is valid only for BDR < 3.6.11, because starting from 3.6.11 the built-in function `bdr.difference_fix_xact_set_avoid_conflict` is used instead.
- Fixed an issue where the `bdr_livecompare` replication origin was being unnecessarily created in BDR databases if automatic fix was disabled.
- Fixed an issue where a comparison worker process might not finish cleanly.
- CI: LiveCompare packages are now also built for Ubuntu 18.04 LTS.

1.2.0 (2020-01-03)

- LiveCompare now supports BLOB fields on Oracle versus Postgres comparison.
- Ignored divergences: 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 (RM8939).
- Volatile divergences: If upon a difference re-check `xmin` has changed on an inconsistent row, then LiveCompare stops re-checking and marks the difference as Volatile (RM10964).
- Overwritten divergences: After the automatic fix was applied, if upon a re-check `xmin` has changed, it means that the row was changed after we have fixed it. LiveCompare marks the divergence as Overwritten (RM10964).
- Unfixable divergences: After the automatic fix was successfully applied, if upon a re-check `xmin` has not changed yet the divergence still remains, LiveCompare marks the divergence as Unfixable (RM10964).
- LiveCompare now returns code = 0 when there are no divergences and return code > 0 when there are divergences.

- Table Filter and Row Filter are now saved in table `<livecompare_schema_name>.settings` , alongside all General Settings.
- New general setting `difference_check_nap_time` , to control how many seconds the difference check worker will sleep before starting a new difference check sprint. Default: 5 seconds.
- When building the table list for BDR 3, LiveCompare now does not consider declarative partitions (RT66502).
- LiveCompare now generates a DML script only for PostgreSQL connections.
- When automatic fix is enabled (`difference_mode = live_fix` or `difference_mode = offline_fix`), it is required that the Right Connection is PostgreSQL.
- Fixed security issue: table/column names and all literals are now properly quoted (RM12530).
- Fixed issue when reserved words were used as column names (RM12530).
- Fixed issue in column names in view `vw_differences` (RM12529).
- Fixed an issue where table was being split unnecessarily in hash compare.

1.1.0 (2019-12-18)

- When building the table list for BDR 3, LiveCompare now considers only the intersection of replication sets that are associated with both BDR nodes from Left Connection and Right Connection (RT66502).
- Fixed issue in building table list when partitioned tables or partitions were being considered (RT66499).
- Improved log verbosity for initial steps of connection validation and building table list.
- Fixed issues with encoding and string handling for Oracle.

1.0.0 (2019-12-03)

- LiveCompare is able to create and use a replication origin in BDR. If BDR version is 3.6.11 or higher, LiveCompare requires an user with `bdr_superuser` permissions or a PostgreSQL superuser to perform replication origin management using BDR functions `bdr.difference_fix_origin_create(text)` , `bdr.difference_fix_session_setup(text)` , `bdr.difference_fix_session_reset()` and `bdr.difference_fix_xact_set_avoid_conflict()` . If BDR version is 3.6.10 or lower, LiveCompare requires a PostgreSQL superuser to perform replication origin management using PostgreSQL functions. Otherwise LiveCompare does not try to manage replication origins. (RT66192 and RM11000).
- LiveCompare is able to fetch replication origin information from each inconsistent row in BDR/pglogical. If BDR version is 3.6.11 or higher, LiveCompare requires an user with `bdr_superuser` permissions or a PostgreSQL superuser to fetch replication origin information from each row. If BDR version is 3.6.10 or lower or pglogical 3 is being used, LiveCompare requires an user with `pglogical_superuser` permissions or a PostgreSQL superuser to fetch replication origin information from each row. Otherwise LiveCompare does not try to fetch replication origin information (RM11971).
- Remove partition from table list if parent table is already on table list (RT65920 and RM10994).
- Always consider replication set tables when building table list for pglogical and BDR (RT65920).
- Fixed issue when handling empty strings (RT65918 and RT65988).
- Fixed issue in upgrading from 0.11.0 to 0.12.0 (RT65918 and RT65988).
- Fixed issue in min PK and max PK value determination.

0.12.0 (2019-11-01)

- Changed default value of setting `difference_fix_start_query` to change role to the owner of the table, in the automatic fix transaction. This is done in order to prevent database users from gaining access to the privileged role used by LiveCompare by writing malicious triggers. As a result the user for the Right Connection needs to have ability to switch role to the table owner (RM11000).
- Handled cases when a table is dropped or receive breaking schema changes after LiveCompare has built the table list and before LiveCompare has started the comparison round on the table. Now LiveCompare checks and updates metadata about the table before the comparison round (RT65918).
- Fixed issue when executing `pg_replication_origin_session_setup` (RT65988).
- Fixed issue in max PK value determination (RM11340).
- Fixed performance issue when fetching metadata from tables, when there is a large number of tables in the database.

0.11.0 (2019-10-10)

- Each difference check is now logged in table `difference_log`, which provides useful information for analysis of each difference as it evolves over time. Each difference can pass through one of the following statuses: - **Detected(D)**: The difference was just detected. If `difference_mode = live_nofix` or `difference_mode = live_fix`, then LiveCompare will re-check the difference multiple times until it comes to a conclusion (see other statuses below), or at maximum N times (configurable via setting `max_difference_check_attempts`), waiting X seconds between each re-check (also configurable via setting `min_time_between_difference_checks`). If `difference_mode = offline_nofix` or `difference_mode = offline_fix`, then each difference found is immediately considered **Permanent**. - **Permanent(P)**: After having re-checked the difference for `max_difference_check_attempts` times, LiveCompare stops re-checking and marks the difference as **Permanent**. If `difference_mode = offline_nofix` or `difference_mode = offline_fix`, then all differences are marked as permanent at the moment they are detected, because there is no re-check. - **Absent(A)**: If before having reached `max_difference_check_attempts`, LiveCompare finds that the difference does not exist anymore (the row is now consistent between both databases), then LiveCompare stops re-checking and marks the difference as **Absent** (in previous versions, LiveCompare would remove the register from the difference table). - **NotAllowed(N)**: The difference was detected, but LiveCompare is forbidden to automatically fix this difference because user has limited the types of differences that can be automatically fixed (via settings `difference_statements` and/or `difference_allow_null_updates`). - **Fixed(F)**: The difference was automatically fixed by LiveCompare in the Right Connection, by applying the DML from field `difference_dml_right`. - **Error(E)**: LiveCompare tried to fix the difference by applying the DML from field `difference_dml_right` against the Right Connection, but got an error. Error message is logged in field `difference_dml_error`.
- Automatic schema changes for Output Connection: if user is pointing Output Connection to a database which was used in previous LiveCompare versions, LiveCompare will automatically handle the schema changes. For 0.11.0, LiveCompare re-creates the schema, but starting from 0.12.0, the implementation will only apply schema changes, keeping user data.
- Created tables `connections` and `settings` to store session values coming from `.ini` file.
- Improved row representation for each difference: whole row is now stored as JSON.
- Storing `xmin` on extended columns of each different row.
- Extended columns (`ctid`, `xmin` and `origin`) of each different row are now also stored as JSON.
- Improved DML for text columns with multi-line values, columns with binary data and array columns.
- Fixed an unhandled exception when table had no PK and columns with multi-line strings, null values or of type `bytea`. Or if table had a PK, but if PK had one of the mentioned situations.

0.10 (2019-09-24)

- Added new setting `difference_mode` which can be: - `offline_nofix` : Tables being compared are not under load, so differences are not re-checked. Differences are reported but not fixed; - `offline_fix` : Tables being compared are not under load, so differences are not re-checked. Differences are reported and fixed in the Right Connection when they are found; - `live_nofix` (default): LiveCompare assumes that tables being compared are under load, so LiveCompare will re-check them to see if they are gone due to eventual consistency. Permanent differences are reported but not fixed. - `live_fix` : LiveCompare assumes that tables being compared are under load, so LiveCompare will re-check them to see if they are gone due to eventual consistency. Differences are reported and fixed when they are marked as permanent.
- Setting `difference_recheck` merged into `difference_mode` . Behavior of `difference_recheck = off` mapped to `difference_mode = offline_nofix` and `difference_recheck = on` mapped to `difference_mode = live_fix` .
- Added new global setting `difference_statements` , which controls what kind of DML statements will be generated by LiveCompare in the DML scripts if `difference_mode = offline_nofix` or `difference_mode = live_nofix` , or automatically applied when `difference_mode = offline_fix` or `difference_mode = live_fix` . The value of `difference_statements` can be: - `all` (default) - `inserts` - `updates` - `deletes` - `inserts_updates` - `inserts_deletes` - `updates_deletes`
- Added new global setting `difference_allow_null_updates` (default `on`), which determines whether commands like `UPDATE SET col = NULL` will be allowed in difference report or automatic fix.
- Added new global setting `difference_fix_replication_origin` , automatically set by default to `bdr_livecompare` for pglogical 3 and/or BDR 3 comparisons if not manually set. LiveCompare will create the specific replication origin in the Right Connection if it doesn't exist, and apply all automatic DML fixes using this replication origin when `difference_mode = live_fix` or `difference_mode = offline_fix` . Note that the replication origin that LiveCompare creates is not dropped to allow verification after the comparison, but if needed the replication origin can then be dropped by using `SELECT pg_replication_origin_drop('<value of setting difference_fix_replication_origin>');` .
- Added new global setting `difference_fix_start_query` , which is executed at the beginning of each transaction to automatically fix differences on the Right Connection. For BDR 3.6.7 and above, if `difference_fix_start_query` is empty, LiveCompare automatically sets `difference_fix_start_query = SET LOCAL bdr.xact_replication = off;` . LiveCompare also automatically sets `difference_fix_start_query` to make the difference fix transaction use the replication origin specified in `difference_fix_replication_origin` .
- Added new Connection setting `start_query` , which can be used to execute any arbitrary query each time a connection is open.
- Added new global setting `show_progressBars` (default `on`), which determines whether or not progress bars should be shown in the console output. Useful for batch executions.
- On Postgres comparisons, each difference found now also stores the `ctid` of the row. If BDR 3 or pglogical 3 is being used, then each difference found also stores the replication origin of the `xmin` of the row.
- Generated DML scripts will always put all DML inside a single transaction. If `difference_fix_start_query` is defined (either manually or automatically), then it is added at the beginning of the transaction.
- Fixed an issue with the global progress bar not being removed at the end of the execution.
- Fixed a bug where Output database existence was not being checked.

0.9 (2019-08-30)

- Support to Oracle databases on Left or Right connections. Oracle Instant Client and Python module need to be installed separately, but are not required for Postgres databases. LiveCompare works without having connectivity to Oracle.
- Currently row hashes and table splitting hashes are only allowed in PostgreSQL versus PostgreSQL comparisons. A new setting `full_comparison_mode` will be automatically set to `on` if a technology other than PostgreSQL is used in any of the connections. If user wants to disable hash usage even on Postgres versus Postgres, `full_comparison_mode = on` can be explicitly defined in the configuration file.
- Support to BDR 1 and 2. When `logical_replication_mode = bdr`, it is possible to define connections with `node_name` and filter tables with `replication_sets`.
- Row hash needs to be `md5()` for both `Left Connection` and `Right Connection` if any of those connections is on PostgreSQL < 11. Otherwise, both connections use `hashtextextended()`. This allows for mixed PostgreSQL version comparison (9.4 versus 12, for example).
- Setting `logical_replication_mode` imposes a validation for PostgreSQL version and extension existence on `Initial Connection` and `Left Connection`. Note that table list is built from `Left Connection`. But on `Right Connection`, only connectivity is checked. This allows for mixed technology comparison (PostgreSQL versus Oracle, BDR versus PostgreSQL, BDR 2 versus BDR 3, etc).
- New setting `difference_recheck` (boolean, default `on`) that allow users to enable or disable difference re-checking.
- Table schema differences (column names and column data types) are logged into the reporting database (table `tables`) for later analysis.
- Left and Right connection information is being logged into the reporting database (table `sessions`) for later analysis.
- Fixed a bug in difference checking in a corner case where tables have some duplicate rows.
- Fixed a bug in difference reporting if a table have more rows in the Right Connection.
- Fixed a bug in difference reporting if there are any temporary differences.

0.8 (2019-08-07)

- Fixed fetching of a single row (to check inconsistency) when table has PK with multiple fields.

0.7 (2019-08-06)

- Fixed handling of tables without rows.
- Better handling of empty sections.

0.6 (2019-06-17)

- Changed logging component.

0.5 (2019-06-17)

- Bug fixes: - Support to sorting data types without ordering operator. - Using `md5()` as a record hash when `hashtextextended()` is not available (PG <= 10). - When configuration file does not exist, show an appropriate message.

0.4 (2019-06-12)

- Preparations for including into 2ndQuadrant CI pipeline.

0.3 (2019-05-29)

- Support to DSN to specify a connection.
- Improved table and row filter.
- Support to different types of logical replication: - native logical replication - pglogical - bdr
- BDR support - Allow user to specify node names for connections. - Allow user to specify replication sets as table filters.
- Created different test scenarios.

0.2 (2019-05-21)

- Improved hash: using `hashtext()` and `hashtextextended()` instead of `md5()` .
- Fetches are performed using prepared statements.

0.1 (2019-05-17)

- Initial support for PostgreSQL.
- Initial implementation of the standalone mode.

1.2.2 Requirements

LiveCompare requires:

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

LiveCompare requires Debian 10+, Ubuntu 16.04+, or CentOS/RHEL/RockyLinux/AlmaLinux 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:
    - 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
```

(continues on next page)

(continued from previous page)

```
- 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/

1.2.3 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	9.4	Data
PostgreSQL	9.5, 9.6, 10, 11, 12, 13 and 14	Data and/or Output
EDB PostgreSQL Extended	9.6, 10, 11, 12, 13 and 14	Data and/or Output
EDB PostgreSQL Advanced (EPAS)	11, 12, 13 and 14	Data and/or Output
pglogical	2 and 3	Initial, Data and/or Output

continues on next page

Table 1 – continued from previous page

Technology	Versions	Connections
BDR	1, 2, 3 and 4	Initial, Data and/or Output
Oracle	11g, 12c, 18c, 19c and 21c	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`.

1.2.4 Command-line Usage

Compare mode

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

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

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

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 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:

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:

```
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:

```
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:

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

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

Dry-run mode

For example 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=liveoutput

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

As the DSN for the `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 let's imagine that only one of the data connections is set correctly. LiveCompare would fail 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 2 data connections available. So you go ahead and fix the Third Connection, but then LiveCompare still fails with:

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

This happens because we have 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 is able to start the comparison.

However, when setting up a comparison from scratch, it's possible to check beforehand whether or not LiveCompare would abort with a configuration error, and further abortion checks are all shown in the order they are performed by LiveCompare.

This is possible with the --dry-run mode, which will:

- Print all execution abortions that would happen due to configuration issues;
- Print the list of connections with some details, including if it's reachable;
- Print the Table Filter;
- Print the list of tables that are common to the reachable connections, after applying the Table Filter.

Here is one sample output, given the INI file exemplified above, 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	BDR Version	Pglogical Version	Initial	
second	postgresql	110015	-	-	False	
first	postgresql	-	-	-	False	
third	postgresql	-	-	-	False	

(continues on next page)

(continued from previous page)

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 - - -
+-----+-----+-----+-----+

1.2.5 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 = $1c1$Games Changed$1c1$ WHERE (category) = 10;
INSERT INTO public.categories (category,categoryname) VALUES (17, $1c1$Test 1$1c1$);
INSERT INTO public.categories (category,categoryname) VALUES (18, $1c1$Test 2$1c1$);

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.

Comparison abortion

Before starting the comparison session, LiveCompare tries all connections. If the number of reachable connections is not at least 2, then LiveCompare aborts the whole session with an appropriate error message. If at least 2 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 with an appropriate error message.

Considering the tables available on all reachable connections, LiveCompare builds the list of tables to be compared, also taking into account the `Table Filter`. If a specific table does not exist on at least 2 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 does not 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 specific table having the error has its table comparison aborted and LiveCompare moves on to the next table.

For each table LiveCompare starts the table comparison, first LiveCompare checks the table definition on all reachable connections. If the tables don’t have the same columns and column data types, LiveCompare applies the `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 be used in the table comparison, following these rules:

- 1- Use the custom `Comparison Key` if the user configured it; or
- 2- Use PK if available; or
- 3- If the table has `UNIQUE` indexes: among the `UNIQUE` indexes that have all `NOT NULL` columns, use the `UNIQUE` index with less columns; or
- 4- If none of the above is possible, try to use all `NOT NULL` columns as a `Comparison Key` (`NULL` columns can also be considered if `ignore_nullable = false`).

If strategies 1 or 4 above are decided to be used as a `Comparison Key`, then LiveCompare also checks for uniqueness on the key. If uniqueness is not possible, then LiveCompare aborts the comparison on that specific table (this behavior can be disabled with `check_uniqueness_enforcement = false`).

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
);
```

1.2.6 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

[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
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
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, 3 and 4;
- 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 rname = 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, BDR 3.7 or BDR 4 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 and BDR 4.

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,
```

(continues on next page)

(continued from previous page)

```

        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` :

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

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

```

(continues on next page)

(continued from previous page)

```
[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 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::timestampz);;

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.

Conflicts Filter

It's also possible to 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

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 4 node versus vanilla Postgres instance;
- Vanilla Postgres instance versus pglogical node.

1.2.7 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
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
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
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
```

(continues on next page)

(continued from previous page)

```
[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
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` <<http://www.oracle.com/technetwork/topics/intel-macsoft-096467.html>>`\_\_`) and extract in `~/lib` ;
- ****Linux**** : Download Oracle Instant Client `32-bit` <<http://www.oracle.com/technetwork/topics/linuxsoft-082809.html>>`\_\_`) `64-bit` <<http://www.oracle.com/technetwork/topics/linuxx86-64soft-092277.html>>`\_\_`) and install it on your system, then set `LD\_LIBRARY\_PATH` ;
- ****Windows**** : Download Oracle Instant Client `32-bit` <<http://www.oracle.com/technetwork/topics/winsoft-085727.html>>`\_\_`) `64-bit` <<http://www.oracle.com/technetwork/topics/winx64soft-089540.html>>`\_\_`) 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 will abort the comparison for that specific table, and you will see the following error message 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`.

The Common Hash uses the `standard_hash` function on Oracle 12c and newer. On Oracle 11g, the `standard_hash` function is not available, so LiveCompare tries to use the `dbms_crypto.hash` function instead, but it might require additional privileges for the user on Oracle side, for example:

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

1.2.8 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` (disabled 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_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 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:

	pre- pared_statements	server_side_cursors_	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

continues on next page

Table 2 – continued from previous page

	pre-prepared_statements	server_side_cursors	server_side_cursors_without_hold
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 fetched from Oracle using a direct query, and 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 will have 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 a large table to be compared in parallel by multiple workers, performance for each worker can be impacted by the hash condition being applied to all rows. Depending on the Postgres configuration (specially 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, and if the database is running on SSD, disabling Bitmap Heap Scan on LiveCompare can significantly improve the comparison performance. This can be done per connection with the `start_query` setting:

```
start_query = set enable_bitmapsan = off
```

- `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_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 only happen during a heart beat, so `min_time_between_round_saves` should be greater than or equal `min_time_between_heart_beats`. 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.

- `comparison_cost_limit` : if > 0, corresponds to a number of rows each worker will process before taking a nap of `comparison_cost_delay` seconds. Defaults to 0, meaning that each worker will process rows without taking a nap.
- `comparison_cost_delay` : if `comparison_cost_limit` > 0, then this setting specifies how long each worker should sleep. Defaults to 0.0.
- `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_progressBars` : 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`.

- `ignore_nullable` : If for a specific table comparison, LiveCompare is using a Comparison Key different than the Primary Key, then LiveCompare requires all columns to be NOT NULL if `ignore_nullable` is enabled (default). It's possible to override that behavior by setting `ignore_nullable = off`, which will allow 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 an 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 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.

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` will cause 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) causes the `Row Filter` to be included in the prepared query, which is then executed at every data buffer that is fetched. Which means that,

if the query uses `now()` , `CURRENT_TIMESTAMP` or `SYSDATE` (on EPAS) on the **Row Filter** , then when the prepared statement is executed, Postgres will re-evaluate the current timestamp.

So, when using `now()` , `CURRENT_TIMESTAMP` or `SYSDATE` on the **Row Filter** , for example:

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

Then it is also required to use a server-side cursor to ensure the current timestamp is evaluated only at the beginning of the queries. In other words, it is required that `data_fetch_mode` is set to a value different than `prepared_statements` .

On Oracle, as explained above, the `data_fetch_mode` setting is ignored, and the query is executed at the beginning, then data is fetch via the client-side cursor. This 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 of using a server-side cursor in Postgres.

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.

Comparison Key

Similarly to the `Column Filter`, in this section you can also specify a list of columns per table. These columns will be 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 the example above, for table `public.table1`, the Comparison Key will be columns `col_a` and `col_b`. For table `public.table2`, columns `c1` and `c2` will be considered as a Comparison Key.

The same behavior about missing columns or filtered out or missing tables that are explained in the `Column Filter` section above, 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 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.

1.2.9 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

This Source Code Form is subject to the terms of the Mozilla Public License, v. 2.0. If a copy of the MPL was not 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.

© Copyright 2016, 2020, Oracle and/or its affiliates. All rights reserved. Portions Copyright © 2007-2015, Anthony Tuininga. All rights reserved. Portions Copyright © 2001-2007, Computronix (Canada) Ltd., Edmonton, Alberta, Canada. All rights reserved Revision 10e5c258.

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

Psycpg2

psycpg2 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.

psycpg2 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 psycpg2 (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 `psycpg/adapter*.{h,c}` and `psycpg/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

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.