Design and Implementation

This document mainly consists of 4 parts, which are:

- Metadata management. Where are history data kept? What is modified to data dictionary?
- Data manipulation. What happens when carrying out DML?
- System running. Purge thread and history data restoration.
- Temporal query. Procedure of temporal query and history data visibility.

Definition

Data statuses are defined as follows:

Current State: Under MVCC or lock-based concurrency control, the latest data version is in current state.

Transitional State: Under MVCC, some active but not the latest transactions are reading a data version, while the latest transaction changes the data, then the version being read is between Current State and Historical State, that is so-called Transactional State. In MySQL/InnoDB, transitional data locate in undo and are to be historical finally.

Historical State: Under MVCC, the data version generated by the transaction prior to the smallest transaction in the active transaction list is a historical version. Under lock-based concurrency control, after a transaction commit, the version before commit is in historical state. In MySQL/InnoDB, historical data locate in undo and are to be purged.

Current Table: A table keeping current state data.

History table: A table keeping historical data.

Restore: A procedure moving historical data from undo to history table.

Metadata Management

For a table *t* with temporal needs, create a table *t_history*. Table *t* is a current table and *t_history* is a history table. DML/DQL/DDL/Purge on current, history and ordinary (not current nor history) tables are different.

SQL

Syntax is extended to create current and history table in a single statement.

```
CREATE TABLE [IF NOT EXISTS] tbl_name (create_definition,...)[table_options][WITH TEMPORAL]
    table_options:
          table_option[[,] table_option] ...
      table_option:
          AUTO_INCREMENT [=] value
        |   ...
        |   HIST_TABLESPACE [=] value
```

- **WITH TEMPORAL**: Denotes a temporal need so that history table is to be created.
- **HIST_TABLESPACE**: Tablespace of history table. Default history tablespace *ibhistory* is used if *HIST_TABLESPACE* is not given.
- *CREATE LIKE*、*CREATE SELECT*、*TEMPORARY*、*PARTITION* are not supported.

High Level Architecture

- As SQL:2011, current table stores current data, history table stores historical versions.
- Data dictionary distinguishes current, history and ordinary tables, so that some constraints can be made, i.e., temporal query doesn’t work on ordinary tables, only old versions of a current table are restored, history tables cannot be accessed directly.
- Insert, update, delete on history table are prohibited for the reason of data protection. Considering the data growth, a super user in socket connection could make delete on history table, while insert and update are still prohibited.
- Direct query on history table is prohibited, use temporal query on current table to access historical versions instead. This is what SQL:2011 regulates.
- DDL on current and history table is prohibited, because a synchronous altering on current and history table is not implemented and altering on each table separately may result in inconsistent schema.
- History and current table get the same schema. Historical data and current data are in the same format and structure, the only difference is lifetime.
- Use SHOW CREATE TABLE, DESCRIBE or query information_schema to see current and history tables’ schemas.
- An extra index on InnoDB system column DB_TRX_ID is created for history table to speed up query in the dimension of transaction id.
- History table is kept in an isolated tablespace so that current and history date can be stored in different storage media, i.e., SSD for current table, HDD for history table.

Low Level Design

Modified code:

Extend CREATE TABLE syntax

sql/sql_yacc.yy, sql/lex.h, sql/gen_lex_token.cc

CREATE TABLE Procedure in SQL Layer

To create current and history tables in a single statement.
sql/sql_table.cc

- Function *mysql_create_table*
  For a create statement with *WITH TEMPORAL*, *mysql_create_table* invokes *mysql_create_table_inner* to create a current table, then *mysql_prepare_create_history_table* to prepare for history table, and *mysql_create_table_inner* to create a history table.
  For a create statement without *WITH TEMPORAL*, *mysql_create_table_inner* is invoked to create an ordinary table.
- Function *mysql_prepare_create_history_table*
  Prepare for creating a history table, i.e., concatenate the history table's name, make a history table *TABLE_LIST* object, etc.
- Function *mysql_create_table_inner*
  MySQL original *mysql_create_table* is divided, and *mysql_create_table_inner* creates a table factually.

Distinguish Current/History/Ordinary Tables

Add fields or flags to distinguish current, history and ordinary tables in dd, table cache and innobase.

sql/dd/types/abstract_table.h

- Class *Abstract_table*
  Add member attributes hist_tbl and orig_tbl of bool type.
  hist_tbl==1&&orig_tbl==0→ history table; hist_tbl==0&&orig_tbl==1→current table;
  hist_tbl==0&&orig_tbl==0→ordinary table; hist_tbl==1&&orig_tbl==1→impossible.

sql/dd/dd_table.cc

- Function *fill_dd_table_from_create_info*
  It is invoked when creating table and calls *Abstract_table::set_hist* or *Abstract_table::set_orig* in accordance with *HA_CREATE_INFO*, which knows whether the create statement contains *WITH TEMPORAL*.

sql/dd/impl/types/abstract_table_impl.cc

- Function *Abstract_table_impl::store_attributes*
  Serialize *Abstract_table.hist_tbl* and *orig_tbl*.
- Function *Abstract_table_impl::restore_attributes*
  Deserialize *Abstract_table.hist_tbl* and *orig_tbl*.

sql/table.h

- Struct *TABLE_LIST*
  Member attributes *hist_tbl* and *orig_tbl* are added. In server layer, use *TABLE_LIST* to see whether it's a current, history or ordinary table.
- Struct *TABLE*
  Member attributes *hist_tbl* and *orig_tbl* are added.

storage/innobase/include/dict0mem.h

- Macro *DICT_TF2_ORIG_TABLE, DICT_TF2_HIST_TABLE*
  Add two bitmasks *DICT_TF2_ORIG_TABLE* and *DICT_TF2_HIST_TABLE*. In InnoDB layer, use *dict_table_t::flags2* to see whether it's a current, history or ordinary table.
  create_table_info_t::innobase_table_flags() sets *flags2* when creating a table.
  dd_fill_dict_table() sets *flags2* when opening a table.

Constraints on Current/History/Ordinary Tables

sql/sql_base.cc

- Function *open_table*
  After MySQL Server bootstraps, for the first time opening the table, metadata is read from dd, and *set_and_check_temporal* is called to set *TABLE_LIST.hist_tbl* and *orig_tbl*. Then *TABLE::init()* sets *TABLE.hist_tbl* and *orig_tbl* in accordance with *TABLE_LIST* object.
  Subsequent opening the same table won't access dd, loads *TABLE* object from table cache instead. Then *set_and_check_temporal* sets *TABLE_LIST.hist_tbl* and *orig_tbl* according to *TABLE* object.
- Function *set_and_check_temporal*
  Called by *open_table*.
  Set *TABLE_LIST.hist_tbl* and *orig_tbl* in accordance with *TABLE* object or dd.
  It also:
  a. Raises error if DDL/DML/DQL accesses history table directly. Super user in socket connection is allowed to delete.
  b. Checks that only select statement could contain temporal hint. (This is better to be done in parsing phase.)
  c. Checks that a temporal query can only work on a current table.

Transaction ID Index on History Table

storage/innobase/handler/ha_innodb.cc

- Function *create_table_info_t::create_table*
  Calls *create_trx_id_index* if it's creating a history table.
- Function *create_trx_id_index*
  Creates a secondary index on InnoDB system column *DB_TRX_ID*.

Display Current/History Mark

SHOW CREATE TABLE, DESCRIBE and information_schema.tables display current and history marks.

sql/sql_show.cc

- Function *store_create_info*
  Appends string *"/*FLASHBACK ORIGINAL*/"* or *"/*FLASHBACK HISTORICAL*/"* to display info according to *TABLE_LIST::orig_tbl* and *hist_tbl*.

sql/dd/impl/tables/tables.cc

- Function *Tables::Tables*
  Add fields *has_history_table* and *is_history_table* by *m_target_def.add_field*.

sql/dd/impl/system_views/tables.cc

- Function *Tables_base::Table_base*
  Add fields *HIST_TABLE* and *ORIG_TABLE* by *m_target_def.add_field*. *HIST_TABLE* is defined as *Tables.is_history_table* and *ORIG_TABLE* is defined as *Tables.has_history_table*.

Data Manipulation

High Level Architecture

- DML syntax stays unchanged.
- DML on history table is prohibited.
- Update and delete on current table make data status transfer from current state to historical state and data version move from current table to undo segment.

Low Level Design

- Some extra operations are made when data transfers from current state to transitional state, see System Column DB_END_TRX_ID.

System running

Purge threads clean up old data versions periodically. We modify the purge procedure to restore the old versions into history table.

Requirements：

- Every version gets restored, no one is lost.
- No duplicated versions in history table.
- Restore current table’s old versions only.

High Level Architecture

Purge threads scan undo segment and clean up historical data periodically.
Some extra steps are attached in purge procedure:

- If the version being purged is from ordinary table, do what purge originally did.
- If the version being purged is from current table, then:
    - ■ If it’s an in-place update undo record, construct the old version with its succeeding version and the undo record, and insert it into history table.
    - ■ If it’s a delete marked undo record, put the delete marked version and all the preceding versions into history table. For example, there are 3 versions, v0 – v1 – v2, v0 and v1 are generated by in-place update and v2 is a delete marked version. When purging v2: (1) v2 is restored; (2) v2 + undo rec1 → v1, v1 is restored; (3) v1 + undo rec0 → v0, v0 is restored.

      The reasons why we restore all preceding versions when a delete marked version is being purged are:

      (a) When a purge thread cleans up several versions with the same primary key(which means the same user record), the latest version is cleaned up first. For example, v0, v1, v2 are 3 versions with the same pk, v0 is the oldest and v2 is the newest, then purge thread cleans up them in the order of v2, v1, v0.

      (b) After a delete marked version is purged, the data vanishes from table, while the in-place update undo records need newer versions to construct the older ones. For example, v2 is cleaned up, then it is undoable to construct v1 by v2 + undo rec1, not even v1 + undo rec0 → v0. So, we have to restore all preceding versions as a delete marked version is purged, or in-place update versions can be lost.

  After old versions are restored into history table, purge threads clean up the data from current table.

Low Level Design

Modified code:

Restore In-Place Updated Version

storage/innobase/row/row0purge.cc

- Function *row_purge_upd_exist_or_extern_func*
  This function cleans up old versions generated by in-place update, and calls *row_purge_his_restore_single_ver* if the undo record belongs to a current table.
- Function *row_purge_his_restore_single_ver*
  Restores a single version to history table.
  - Makes a *que_thr_t* object for insert on history table by calling *row_purge_his_restore_prepare_thr*.
  - Copies the current version *rec* to *prev_vers* with *rec_copy*.
  - *row_upd_rec_in_place* is called to backtrack *prev_vers* to the previous version, using *purge_node_t::update*.
  - Opens history table with *row_purge_his_restore_open_table*.
  - *row_ins_clust_index_entry* and *row_ins_sec_index_entry* are called to put the old versions into history table.
  - *trx_commit* and *trx_free_for_background* end the insert transaction on the history table.
  - Closes history table with *row_purge_his_restore_close_table*.

Restore Delete-Marked Version

storage/innobase/row/row0purge.cc

- Function *row_purge_remove_clust_if_poss_low*
  Cleans up delete marked version. If the delete marked version belongs to a current table, *row_purge_his_restore_multi_vers* is called to restore all the preceding versions to history table.
- Function *row_purge_his_restore_multi_vers*
  - Makes a *que_thr_t* object for insert on history table by calling *row_purge_his_restore_prepare_thr*.
  - Opens history table with *row_purge_his_restore_open_table*.
  - Backtracks the version chain:
    - Constructs the prior version with *trx_undo_prev_version_build*.
    - Inserts the prior version into history table with *row_ins_clust_index_entry* and *row_ins_sec_index_entry*.
  - *trx_commit* and *trx_free_for_background* end the insert transaction on the history table.
  - Closes history table with *row_purge_his_restore_close_table*.
- Function *row_purge_his_restore_prepare_thr*
  A *que_thr_t* object, with a *trx_t* object in it, is set up for insert operation.
- Function *row_purge_his_restore_open_table*
  Calls *dd_table_open_on_name* and *lock_table* to open and lock history table.
- Function *row_purge_his_restore_close_table*

Calls *dd_table_close* to close history table as restoration finishes.

Temporal query

Temporal query retrieves current, transitional and historical data.

SQL

Extend syntax to support temporal query.

```
SELECT
        [ALL | DISTINCT | DISTINCTROW] ...
        select_expr [, select_expr ...]
        [FROM table_references [WHERE where_condition] ...]

        table_references:
                tbl_name [[AS] alias] [index_hint_list] [temporal_hint]

        temporal_hint:
                SYSTEM TIME AS OF value [ONLY HISTORY]
              | SYSTEM TIME FROM value1 TO value2 [ONLY HISTORY]
              | SYSTEM TRANSACTION value [ONLY HISTORY]
```

- *tbl_name* is a current table.
- *temporal_hint* is only supported in select. Update and delete with *temporal_hint* raise error
- *ONLYT HISTORY* queries history table only, and current table is skipped.
- Illustrate the semantics of *temporal_hint* with the diagram below.
  A sketch diagram is given for the purpose of illustration. It is not what MySQL Client shows.

Pk	Version	Begin Tid	End Tid	Begin ts	End ts
1	V0	2000	2003	T0	T1
1	V1	2003	2004	T1	T3
1	V2	2004		T3	

  Pk represents a user record.
  Version represents a unique data version of the user record.
  Begin Tid and *End Tid* are the transactions which created and removed the data version.
  Begin ts and *Edn ts* are the timestamp when *Begin Tid* and *End Tid* were committed.

■ AS OF value. A time-point query retrieves the data version which was/is in current state at the *value* time. For example,
*select * from t system time as of T2 where pk=1* gets V1
*select * from t system time as of T1 where pk=1* gets V1. V0 is a history version at T1.

- FROM value1 TO value2. A time-range query retrieves the data versions was/is in current state during *value1~value2* time. It requires a data version's lifetime overlapping the given time range.
  *select * from t system time from T0 to T2 where pk=1* gets V0 and V1， because [V0.BeginTs, V0.EndTs) ∩ [T0, T2] ≠ø， [V1.BeginTs, V1.EndTs) ∩ [T0, T2] ≠ø
- TRANSACTION value. Transaction ID query retrieves the data versions created or removed by this transaction.
  *select * from t system transaction 2003 where pk=1* gets V0 and V1. Transaction 2003 removed V0 and created V1.

High Level Architecture

- Historical data visibility check.
  Identifying when a version was created and removed is the key to check history visibility.
  - System columns – Version's lifecycle in the dimension of transaction ID
    InnoDB system column DB_TRX_ID represents the transaction that created the data version. An extra column DB_END_TRX_ID is introduced to denote the transaction that removed the data version, i.e., DB_TRX_ID is an insert transaction and DB_END_TRX_ID is a delete one.
    DB_END_TRX_ID is UINT64_MAX as a data version is newly inserted.
    DB_END_TRX_ID of an old version is written into undo log as the version is removed by an update or delete. When doing purge or temporal query, DB_END_TRX_ID is extracted from the undo log and written to the old version.
    DB_TRX_ID and DB_END_TRX_ID denote a version's lifecycle in the dimension of transaction ID. Besides, a correspondence between transaction ID and physical time is needed, or it'll be hard to understand the lifecycle.
  - Transaction Status Management – A correspondence between transaction ID and physical time
    Transaction Status Management maintains ID, status, start time and finish time of a transaction. The status is one of {INPROGRESS, COMMITTED, ABORTED, UNDO}. INPROGRESS, COMMITTED and ABORTED are as their literal meanings. UNDO means a user transaction that has not been started or an InnoDB internal transaction, which occupies an id but is not maintained by our system.
    A transaction log record is 15Byte with 1Byte for status, 7Byte for start time and 7Byte for finish time. Sizes of a transaction log file and a page are predefined and constant, then transaction ID and {file number, page number, in-page offset} share a one-to-one mapping.
    When a transaction begins, INPROGRESS and start time are logged. When a transaction finishes, COMMITTED or ABORTED and finish time are logged.
    Finish time matters in history visibility check, cause a committed transaction did create or remove a data version.

  Check whether a version is visible in following steps:

1. Get *DB_TRX_ID* and *DB_END_TRX_ID* fields and find out the committed time of them. The physical times when the version is created and removed, we call them *begin_ts* and *end_ts*, are determined.
2. For an *AS OF value* query, a version is visible if *begin_ts <= value < end_ts*.
3. For a *FROM value1 TO value2* query, a version is visible if *begin_ts <= value2 && end_ts > value1*.
4. For a *TRANSACTION value* query, a version is visible if *DB_TRX_ID==value || DB_END_TRX_ID==value*.

- Concurrency between purge and query

  Temporal query in three stages: 1. Get current version from current table. 2. Get transitional and historical versions from undo segment. 3. Get historical versions from history table.

  A query thread can not access an undo record being purged, which may lead to a missing version and incomplete result set. For example, there is a version chain v0 – v1 – v2, among which v0 is a historical version in history table, v1 is a historical version in undo segment and v2 is the current version in current table. A temporal query got v2 from current table and is trying to get a prior version from undo segment, but v1 cannot be read because it's being purged at the meantime. The temporal query heads to history table then. When the history table is queried, it happens that v1 hasn't been restored yet, then only v2 is returned. Finally, the temporal query got an incomplete result set with v1 being missing.

  To solve this, we add an MDL to control the concurrency between purge and temporal query, with the cost of purge efficiency. We haven't tested specifically how bad purge is affected, but it's definite that the more versions there are, the more time temporal query costs, the slower purge carries out.

- Query transaction status
  - TRXTOTIME(val1, val2, val3)

    val1 is transaction id in string format. val2 and val3 are decimals.

    Get the statuses of transactions whose ids are from [val1-val2, val1+val3], i.e., TRXTOTIME("2003", 1, 1) gets transaction 2002, 2003, 2004, and 2005.
  - TIMETOTRX(val1, val2, val3)：

    val1 is of type datetime. val2 and val3 are decimals in seconds.

    Get the statuses of transactions which are committed during [val1-val2, val1+val3], i.e., TIMETOTRX("2020-01-01 00:00:00", 60, 60) gets transactions which are committed from 2019-12-31 23:59:00 to 2020-01-01 00:01:00.
  - CURRENTTRX()：Get the max transaction ID for now.

Low Level Design

Modified code：

Extend syntax

sql/sql_yacc.yy, sql/lex.h

Extending Server with Temporal Ability

a) Get the type of temporal query, including *as of*, *from to* and *transaction id*, and pass it to InnoDB.
b) Constraints: Only SQLCOM_SELECT on current table works.
c) Query current and history tables in turn.

Temporal Hint

sql/table.h

- Class Temporal_hint
  During parsing, makes a *temporal_hint* object, denoting temporal type and value, for the *TABLE_LIST* object. *TABLE_LIST.temporal_hint* is null for a non-temporal query.

sql/sql_class.cc

- Function *thd_get_table_temporal_hint*
  Passes *temporal_hint* to InnoDB from Server.

Open Current and History Tables

A temporal query on a current table opens current and history tables.

sql/sql_select.cc

- Function *Sql_cmd_dml::prepare*
  Calsl *Sql_cmd_dml::prepare_tempral* to open history table for a temporal query. *Sql_cmd_select::prepare_temporal* calls *dml_prepare_temporal*, while *prepare_temporal* of other commands does nothing.
- Function *dml_prepare_temporal*
  Traverses the queried tables, and calls *make_his_table_list* to add a history *TABLE_LIST* object into *LEX.query_tables* if a current table is found. Tables in *LEX.query_tables* are to be opened.

Constraint

sql/sql_base.cc

- Function *set_and_check_temporal*
  Called by *open_table*.
  TABLE_LIST.temporal_hint denotes whether it's a temporal query. Only *SQLCOM_SELECT* on a current table is allowed for a temporal query.

Query Current and History Tables in Order

include/my_base.h
- Macro *HA_END_OF_ORIG_SCAN*
  A newly introduced error code.
  After a temporal query finishes on the current table, *HA_END_OF_ORIG_SCAN* is returned instead of *HA_ERR_END_OF_FILE*, so that temporal query turns to history table.

sql/sql_executor.cc
- Function *sub_select*
  sub_select gets *HA_END_OF_ORIG_SCAN* and turns to history table.

Extending InnoDB with Temporal Ability

a) Visibility check for temporal query.
b) Transaction status management.

Temporal Query Hint

storage/innobase/include/row0mysql.h
- Struct *row_prebuilt_t*
  - We add additional member *Temporal_hint* t_hint*. After function *build_template* builds *row_prebuilt_t*, function *thd_get_table_temporal_hint* will obtain *Temporal_hint* and store in *row_prebuilt_t.t_hint*
  - We add additional member *n_transitional_vers_fetched*, which will be used in function *row_vers_build_for_flashback_range_read*

Visibility Check for Temporal Query

storage/innobase/handler/ha_innodb.cc
- Function *index_read* and *general_fetch*
  We use the bitmasks, *dict_table_t.flags2 & DICT_TF2_ORIG_TABLE* and *dict_table_t.flags2 & DICT_TF2_HIST_TABLE*, to check whether we are now querying a current table or a history table, the struct *row_prebuilt_t.t_hint* is used to check whether this query is temporal.

We conduct the temporal query with the following steps:

- ■ Query current table
    - ◆ *row_search_mvcc* queries the current table.
    - ◆ Current table is skipped if it's an *ONLY HISTORY* temporal query.
- ■ Query history table
    - ◆ *open_his_dict_table* opens history table
    - ◆ *prepare_prebuilt_before_his_scan* modifies *m_prebuilt* so that it searches history table.
    - ◆ *row_search_mvcc* queries history table.
    - ◆ *reset_prebuilt_after_his_scan* resets *m_prebuilt*.
    - ◆ *close_his_dict_table* closes history table.

- Function *open_his_dict_table*
  Gets history table's name by splicing current table's name and *HISTORY_TABLE_POSTFIX*, and calls function *dd_table_open_on_name* to open the history table.
- Function *prepare_prebuilt_before_his_scan*
  m_prebuilt is a member of *ha_innobase* and is used to retrieve records.
  ha_innobase object of current table is used to query both current and history table. To search history table correctly, *prepare_prebuilt_before_his_scan* adapts *ha_innobase::m_prebuilt* to the history table. After query completion, *reset_prebuilt_after_his_scan* resets *ha_innobase::m_prebuilt*.

storage/innobase/row/row0sel.cc

- Function *row_search_mvcc*
  row_prebuilt_t.t_hint knows the type of temporal query:
    - ■ AS OF
        - ◆ For history table, call function *lock_clust_rec_flashback_point_read_sees* to fetch the visible version.
        - ◆ For current table, call *lock_clust_rec_flashback_point_read_sees* to check whether the latest version is visible, if not, call function *row_sel_build_prev_vers_for_flashback_point* to backtrack previous versions.
    - ■ FROM TO
      Call function *row_sel_build_prev_vers_for_flashback_range* to retrieve multiple visible versions, if there are no more visible versions, goto next_rec.
    - ■ TRANSACTION ID
      Call function *row_sel_build_prev_vers_for_flashback_trx_id* to retrieve multiple visible versions, if there are no more visible versions, goto next_rec.
- Function *row_sel_build_prev_vers_for_flashback_point*
  Call function *row_vers_build_for_flashback_point_read* to obtain visible versions from undo segment.
- Function *row_vers_build_for_flashback_point_read*
  Backtrack the version chain till a visible version is found or there are no older versions.
    - ■ Call function *trx_undo_prev_version_build* to construct a previous version *prev_vers*.
    - ■ Call functions *row_get_rec_trx_id* and *row_get_rec_end_trx_id* to get *prev_vers*' *DB_TRX_ID* and *DB_END_TRX_ID*, then call function *read_tlog_by_trx_id* to get the

commit time of *DB_TRX_ID* and *DB_END_TRX_ID* respectively. See the temporal query High Level Architecture for visibility check.

- Function *row_sel_build_prev_vers_for_flashback_range*
  Calls *row_vers_build_for_flashback_range_read* to retrieve visible versions from undo segment.
- Function *row_vers_build_for_flashback_range_read*
  - *FROM TO* query may return multiple versions for a single primary key, and these versions are kept in a result set cache. The number of versions maybe exceed the capacity of the cache, so that fractional backtrack is possible. To avoid retrieve the same version more than once, a member *n_transitional_vers_fetched* is introduced. n_transitional_vers_fetched denotes the number of versions that have been examined, so that the later backtrack won't bother with the processed versions.
    For example: The capacity of cache is 3. There are versions v0~v5, the first time v3~v5 are cached, and *n_transitional_vers_fetched* is set to 3. The second time, v3-v5 are skipped and v0~v2 are cached. Afterall, v0~v5 are returned to client.
  - Backtrack the version chain till there is no older versions or the result cache is full. Append the visible versions to result set cache.
- Function *row_sel_build_prev_vers_for_flashback_trx_id*
  Calls *row_vers_build_for_flashback_trx_id_read* to get versions created or removed by the given transaction ID from undo segment.
- Function *row_vers_build_for_flashback_trx_id_read*
  - Backtrack the version chain till there is no older version or the cache is full. Append the versions operated by the given transaction ID to result set cache.
- Function *row_sel_flashback_cache_mysql_rec*
  Cache versions in *handler::m_record_buffer* which is to be returned to Server layer.

storage/innobase/lock/lock0lock.cc

- Function *lock_clust_rec_flashback_point_read_sees*
  Called by function *row_search_mvcc* during the AS OF query to check whether the latest data version is visible.
  Calls function *row_get_rec_trx_id* and *row_get_rec_end_trx_id* to get *DB_TRX_ID* and *DB_END_TRX_ID*, and then calls function *read_tlog_by_trx_id* to get the commit time of *DB_TRX_ID* and *DB_END_TRX_ID* respectively. See High Level Architecture for visibility check.

System Column DB_END_TRX_ID

storage/innobase/include/data0type.h

- Macro DB_END_TRX_ID
  DB_END_TRX_ID is appended after DB_ROLL_PTR.

storage/innobase/dict

- Function *dict_table_add_system_columns*

Creates a DB_END_TRX_ID system column when building a table. This field is unnecessary for ordinary table but is still introduced to keep the system columns consistent.

storage/innobase/row/row0ins.cc

- Function *row_ins_step*

  DB_END_TRX_ID is writen as UINT64_MAX as a new version generated.

storage/innobase/trx/trx0rec.cc

- Function *trx_undo_page_report_modify*

  Update and delete write undo by calling *trx_undo_page_report_modify*. The DB_END_TRX_ID field of the undo record is written as the update or the delete transaction ID at this moment and is to be read to construct a prior version later.

Transaction Status Management

storage/innobase/include/tlog0tlog.h

storage/innobase/tlog/tlog0tlog.cc

- Macros
  - TLOG_SIZE_BYTE: size of a transaction status log record
  - PAGE_SIZE_BYTE: size of a physical page
  - FILE_SIZE_BYTE: size of a transaction status log file
  - TLOG_NUM_PER_PAGE: the number of transaction status logs in a page, equals to PAGE_SIZE_BYTE / TLOG_SIZE_BYTE

  We define transaction ID of the first log (in the first page of the first file) as 0. Then we have a one-to-one correspondence between transaction ID and {file number, page number, in-page offset}.

  file no = transaction id / (FILE_SIZE_BYTE / TLOG_SIZE_BYTE)

  page no = (transaction id % (FILE_SIZE_BYTE / TLOG_SIZE_BYTE)) / PAGE_SIZE_BYTE

  page offset = (transaction id % (FILE_SIZE_BYTE / TLOG_SIZE_BYTE)) % PAGE_SIZE_BYTE
- Interfaces
  - Function *record_tlog_low*

    Write transaction status log when transaction starts, commits, and rollback.
  - Function *record_tlog_write_log*

    Write redo log ahead of writing transaction status log.

storage/innobase/include/tlog0lru.h

storage/innobase/tlog/tlog0lru.cc

- The transaction log page forms an LRU List, of which the size is LRU_CACHE_SIZE

Querying Transaction Status

storage/innobase/handler/ha_innodb.cc

- Interface *trxtotime*

  Call function *read_tlog_by_trx_ids* to obtain transaction status logs by transaction IDs.

- Interface *timetotrx*
  Call function *read_tlog_finish_between* to obtain transaction status logs by a specific time interval.
- Interface *currenttrx*
  Call function *trx_sys_get_max_trx_id* to get the max transaction ID for now.

MTR Test

MTR Test Command just as following:

```
./mtr --suite=main --retry-failure=1 --force --max-test-fail=3000
```

Native unit test pass rate (main and suites prefixed by innodb only):

Main	Innodb	Innodb_fts	Innodb_gis	Innodb_undo	Innodb_zip
Failed 261/878 tests, 70.27% were successful	Failed 119/335 tests, 64.48% were successful	Failed 9/46 tests, 80.43% were successful	Failed 11/27 tests, 59.26% were successful	All 5 tests were successful	Failed 11/20 tests, 45.00% were successful

The reason for the failure of the above use cases:
Modifications of the newly introduced system column *DB_END_TRX_ID* may be inadequate, resulting in the core or metadata inconsistency in the testcases. Working on it.

Flashback testcases: mysql-test/suite/flashback

Flashback	Flashback_info	Fselect, Fselect2, Fselect_complex	Restore	Trx_id_index
Test syntax	Test the DDL operations	Test temporary query	Test historical data restoration	Test secondary index on the history table

The pass rate of the additional flashback testcases is taken as 100%.
Testcases would always fail for *Result Mismatch* because the SQLs contain automatically generated timestamps, which are different every time we run mtr, while the query result sets are correct.

Performance Evaluation

Evaluation Environment

CPU	Type	Intel(R) Xeon(R) Platinum 8276 CPU @ 2.20GHz
	Logical Cores	112
Memory	Total Size	990G：Intel AEP – 128G*8, DDR4 2666 – 16G*12
	Swap	0G
Disk	/dev/sda	Intel S4510 – 480G*1

	/dev/nvme0	Intel NVMe – 3.2T*1
Network	Speed	25000Mb/s
	Latency	0.030ms
OS	[root@master ~]# lsb_release –a LSB Version: :core-4.1-amd64:core-4.1-noarch Description: CentOS Linux release 7.2 (Final)	

Evaluation tool

Sysbench 0.5

Parameters

oltp-table-size=2180000
oltp-tables-count=10
max-time=60

Evaulation Results

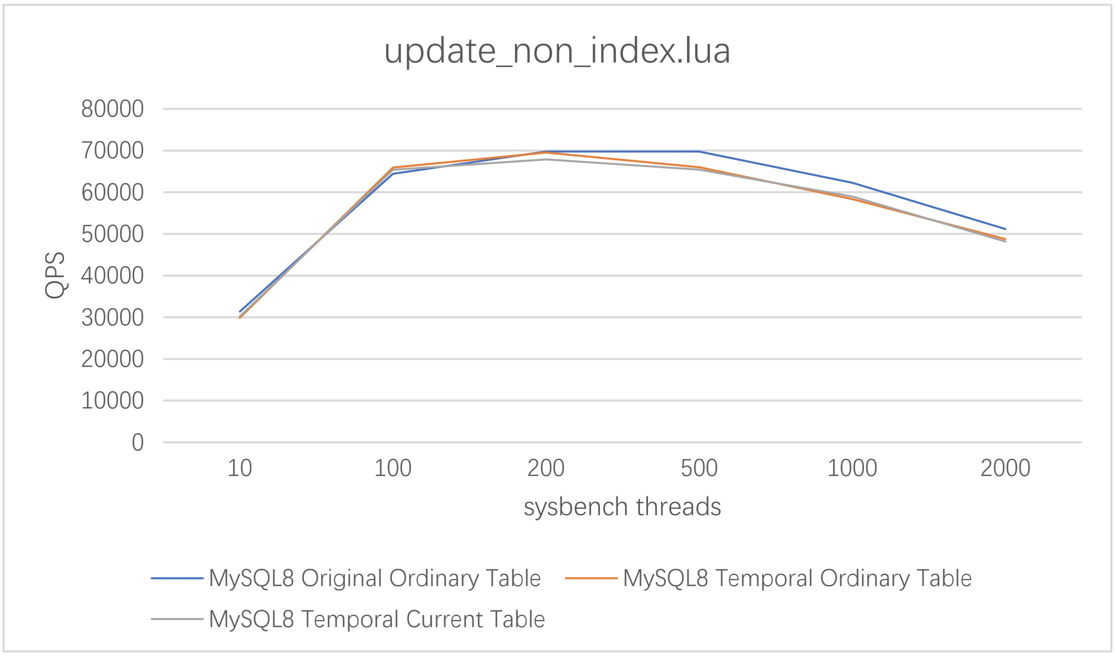
Update_non_index.lua

Query:

UPDATE " .. table_name .. " SET c='" .. c_val .. "' WHERE id=" .. sb_rand(1, oltp_table_size)

Results:

Client Connections	10	100	200	500	1000	2000
MySQL8 Original Ordinary Table	31361.73	64415.19	69765.94	69751.01	62251.56	51178.49
MySQL8 Temporal Ordinary Table	29872.32	65896.73	69508.16	65984.53	58351.66	48745.84
Lost Ratio	4.75%	-2.30%	0.37%	5.40%	6.26%	4.75%
MySQL8 Temporal Current Table	30209.71	65364.77	67867.36	65403.78	58977.45	48149.8
Lost Ration	3.67%	-1.47%	2.72%	6.23%	5.26%	5.92%



Analysis:

1. After introducing the flashback function, the update operation on the ordinary table (without *WITH TEMPORAL* keywords when being created) introduces up to 6% performance overhead.
   a) The performance loss is mainly caused by the transaction status management.
   b) It's considerable to optimize the transaction status management and adopt a lock-free LRU strategy to save the overhead.
2. After introducing the flashback function, the update operation on the flashback table (with *WITH TEMPORAL* keywords when being created) introduces a maximum of 6% performance overhead.
   a) For the same reason as above.

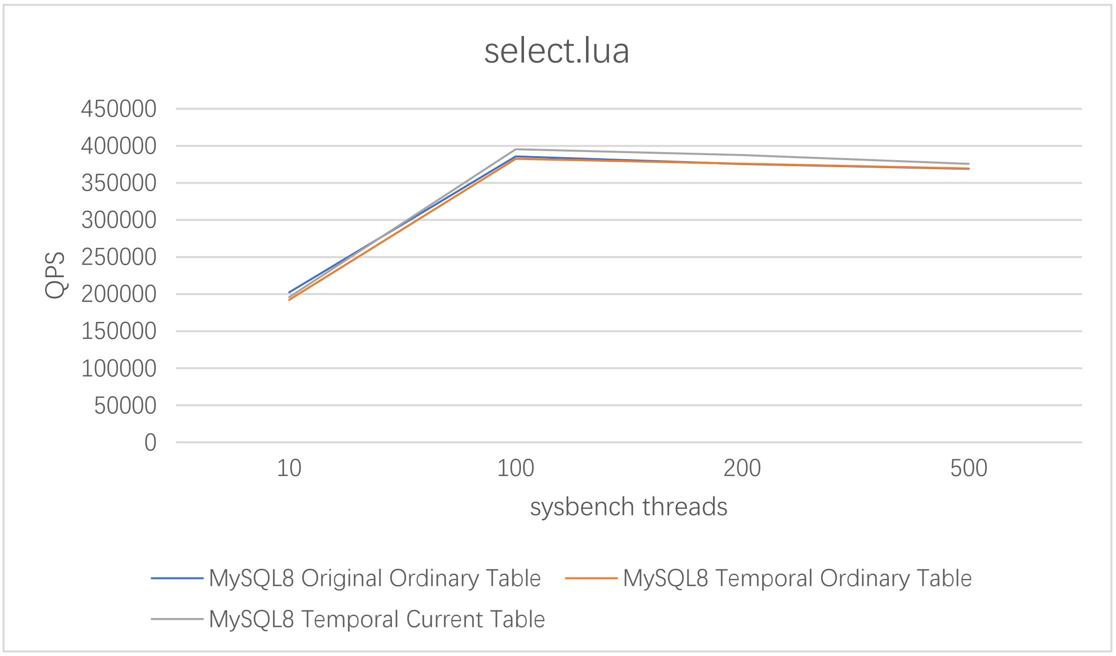
Select.lua

Query:

SELECT pad FROM ".. table_name .." WHERE id=" .. sb_rand(1, oltp_table_size)

Results:

Client Connections	10	100	200	500
MySQL8 Original Ordinary Table	202398.12	385787.4	375499.25	369127.8
MySQL8 Temporal Ordinary Table	192253.8	382550.47	375894.38	369371.19
MySQL8 Temporal Current Table	195818.31	395471.94	387619.93	375790.26



Analysis:
No performance overhead is introduced.

Flashback.lua
Query:
AS OF Query:
SELECT pad FROM ".. table_name .." SYSTEM TIME AS OF '2020-03-23 15:40:30' WHERE id=" .. sb_rand(1, oltp_table_size)

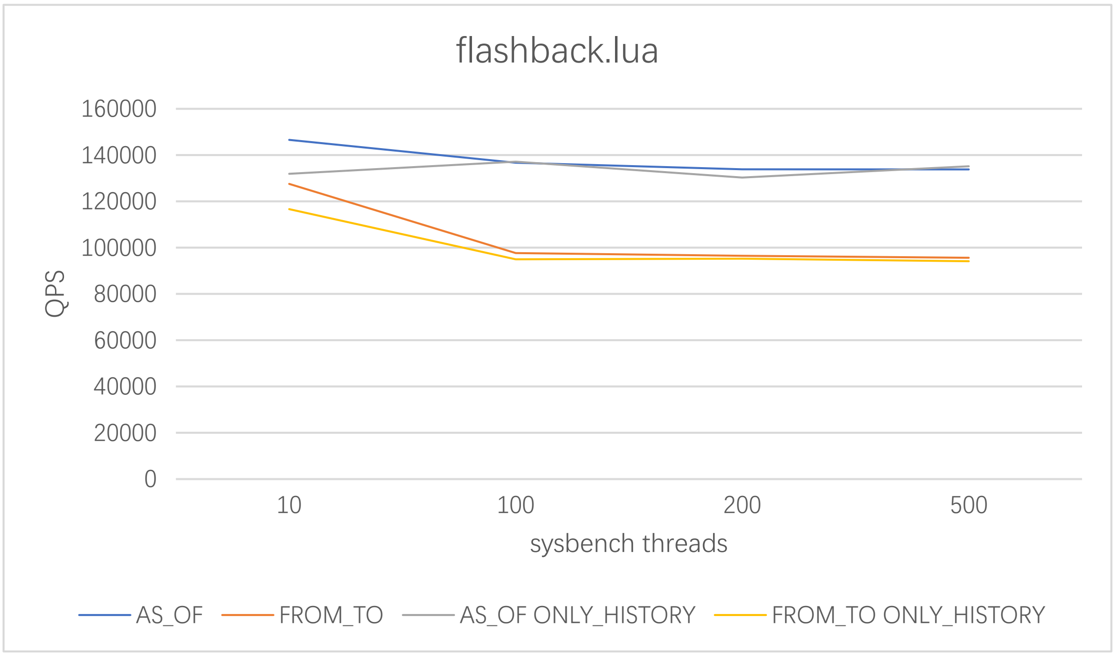
FROM TO Query:
SELECT pad FROM ".. table_name .." SYSTEM TIME FROM '2020-01-01 00:00:00' TO '2021-01-01 00:00:00' WHERE id=" .. sb_rand(1, oltp_table_size)

AS_OF ONLY_HISTORY Query:
SELECT pad FROM ".. table_name .." SYSTEM TIME AS OF '2020-03-22 15:40:30' ONLY HISTORY WHERE id=" .. sb_rand(1, oltp_table_size)

FROM_TO ONLY_HISTORY Query:
SELECT pad FROM ".. table_name .." SYSTEM TIME FROM '2019-01-01 00:00:00' TO '2020-01-01 00:00:00' ONLY HISTORY WHERE id=" .. sb_rand(1, oltp_table_size)

Results:

Client Connections	10	100	200	500
AS_OF	146584.57	136679.75	133847.66	133819.41
FROM_TO	127565.85	97674.83	96493.24	95640.6
AS_OF ONLY_HISTORY	131939.14	137169.13	130304.69	135173.34
FROM_TO ONLY_HISTORY	116660.24	94984.7	95277.84	94149.06



The main performance costs of flashback query are:

a) Obtaining the transaction information from the transaction status log. This can be reduced by optimizing the transaction log.

b) History version traversal overhead. We have made certain optimizations through the secondary index on DB_TRX_ID, additionally, we can introduce a targeted index strategy to optimize the search process of historical versions.

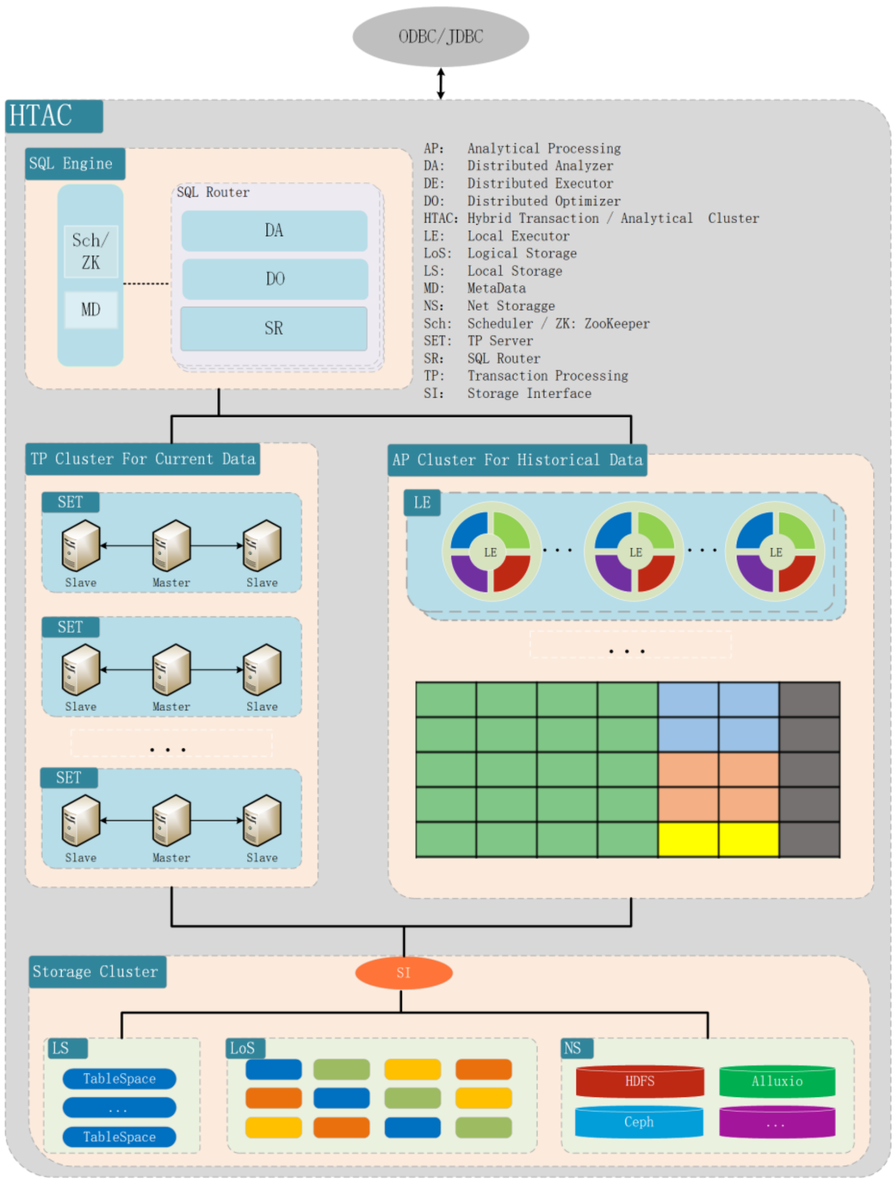
Remaining Problem

- Introducing system column *DB_END_TRX_ID* results in mtr failures.

Future Work

Current data are critical for OLTP scenarios, while historical data are useful for OLAP scenarios. Based on our implementation, the granularity of historical data is much smaller while the accuracy is higher, which has advantages in analytical processing. The current and historical data could be stored in different storage engines to facilitate the targeted optimizations of OLTP and OLAP respectively, we call this extension as Hybrid Transaction and Analytical Cluster (HTAC).

Zoom in HTAC, OLTP cluster is responsible for transactional business, while OLAP system handles analytical business, like historical data processing. According to the semantics of query statements and query operations, queries are sent to the specific cluster for processing through the unified routing module. Considering that OLAP processing takes up a lot of resources, the separating design of HTAC could minimize the impact to the production OLTP system. Additionally, since the magnitude of historical data is quite large, HTAC is quite suitable to achieve unlimited storage of historical data by expanding the volume of storage by HDFS and stuff.



Looking forward to a long-term communication and contribution.