MaterializeMySQL Database engine in ClickHouse

WinterZhang(张健)

About me

- Active ClickHouse Contributor
 - MaterializeMySQL Database Engine
 - Custom HTTP Handler
 - MySQL Database Engine
 - BloomFilter Skipping Index
 - Query Predicate Optimizer
 - And more ~ 400+ commit



https://github.com/zhang2014

MySQL Table Engine

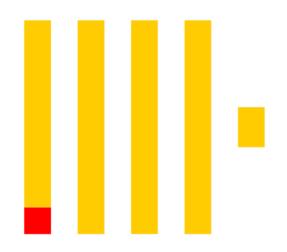
- Mapping to MySQL table
- Fetch table struct from MySQL
- Fetch data from MySQL when execute query

MySQL Database Engine

- Mapping to MySQL database
- Fetch table list from MySQL
- Fetch table struct from MySQL
- Fetch data from MySQL when execute query

- Mapping to MySQL database
- Consume MySQL BINLOG and store to MergeTree
- Experimental feature (20.8, recommend latest stable version)







History data

New data





Check MySQL status

Select history data





Check MySQL status

Select history data



Check MySQL status





Check MySQL status

Select history data





```
connection = pool.get();
            MaterializeMetadata metadata(connection,
getDatabase(database name).getMetadataPath() +
"/.metadata", mysql database name, opened transaction, mysql version);
            if (!metadata.need dumping tables.empty())
                Position position;
                position.update(metadata.binlog position,metadata.binlog file,
metadata.executed_gtid_set);
                metadata.transaction(position, [&]()
                    cleanOutdatedTables(database_name, global_context);
                    dumpDataForTables(connection, metadata, query prefix,
database name, mysql database name, global context, [this] { return
isCancelled(); });
                });
            }
            connection->query("COMMIT").execute();
```



```
connection = pool.get();
            MaterializeMetadata metadata(connection,
getDatabase(database name).getMetadataPath() +
"/.metadata", mysql database name, opened transaction, mysql version);
            if (!metadata.need dumping tables.empty())
                Position position;
                position.update(metadata.binlog position, metadata.binlog file,
metadata.executed_gtid_set);
                metadata.transaction(position, [&]()
                    cleanOutdatedTables(database_name, global_context);
                    dumpDataForTables(connection, metadata, query prefix,
database name, mysql database name, global context, [this] { return
isCancelled(); });
            connection->query("COMMIT").execute();
```



```
connection->query("FLUSH TABLES;").execute();
    connection->query("FLUSH TABLES WITH READ LOCK;").execute();

    fetchMasterStatus(connection);
    connection->query("SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE
READ;").execute();
    connection->query("START TRANSACTION /*!40100 WITH CONSISTENT SNAPSHOT
*/;").execute();

    opened_transaction = true;
    need_dumping_tables = fetchTablesCreateQuery(connection, database,
fetchTablesInDB(connection, database));
    connection->query("UNLOCK TABLES;").execute();
```





```
connection = pool.get();
            MaterializeMetadata metadata(connection,
getDatabase(database name).getMetadataPath() +
"/.metadata", mysql database name, opened transaction, mysql version);
            if (!metadata.need dumping tables.empty())
                Position position;
                position.update(metadata.binlog position, metadata.binlog file,
metadata.executed_gtid_set);
                metadata.transaction(position, [&]()
                    cleanOutdatedTables(database_name, global_context);
                    dumpDataForTables(connection, metadata, query prefix,
database name, mysql database name, global context, [this] { return
isCancelled(); });
                });
            connection->query("COMMIT").execute();
```



Select history data



}

```
/// ISV format metadata file.
writeString("Version:\t" + toString(meta_version), out);
writeString("\nBinlog File:\t" + binlog_file, out);
writeString("\nExecuted GTID:\t" + executed_gtid_set, out);
writeString("\nBinlog Position:\t" + toString(binlog_position), out);
writeString("\nData Version:\t" + toString(data_version), out);

out.next();
out.sync();
out.close();
}
commitMetadata(std::move(fun), persistent_tmp_path, persistent_path);
```



```
connection = pool.get();
            MaterializeMetadata metadata(connection,
getDatabase(database name).getMetadataPath() +
"/.metadata", mysql database name, opened transaction, mysql version);
            if (!metadata.need dumping tables.empty())
                Position position;
                position.update(metadata.binlog position, metadata.binlog file,
metadata.executed_gtid_set);
                metadata.transaction(position, [&]()
                    cleanOutdatedTables(database_name, global_context);
                    dumpDataForTables(connection, metadata, query_prefix,
database name, mysql database name, global context, [this] { return
isCancelled(); });
            connection->query("COMMIT").execute();
```



```
auto iterator = master_info.need_dumping_tables.begin();
    for (; iterator != master_info.need_dumping_tables.end() && !

is_cancelled(); ++iterator)
    {
        const auto & table_name = iterator->first;
        tryToExecuteQuery(query_prefix + " " + iterator->second,
query_context, database_name, comment); /// create table.

        auto out =
std::make_shared<CountingBlockOutputStream>(getTableOutput(database_name,
table_name, query_context));

        MySQLBlockInputStream input(connection, "SELECT * FROM " +
backQuoteIfNeed(mysql_database_name) + "." + backQuoteIfNeed(table_name),
out->getHeader(), DEFAULT_BLOCK_SIZE);

        Stopwatch watch;
        copyData(input, *out, is_cancelled);
}
```





```
auto iterator = master_info.need_dumping_tables.begin();
    for (; iterator != master_info.need_dumping_tables.end() && !
is_cancelled(); ++iterator)
    {
        const auto & table_name = iterator->first;
        tryToExecuteQuery(query_prefix + " " + iterator->second,
query_context, database_name, comment); /// create table.

        auto out =
std::make_shared<CountingBlockOutputStream>(getTableOutput(database_name,
table_name, query_context));

        MySQLBlockInputStream input(connection, "SELECT * FROM " +
backQuoteIfNeed(mysql_database_name) + "." + backQuoteIfNeed(table_name),
out->getHeader(), DEFAULT_BLOCK_SIZE);

        Stopwatch watch;
        copyData(input, *out, is_cancelled);
}
```

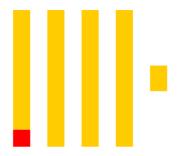




```
NamesAndTypesList columns name and type = getColumnsList(create defines-
>columns);
    columns->set(columns->columns,
InterpreterCreateQuery::formatColumns(columns_name_and_type));
    columns->columns-
>children.emplace back(create materialized column declaration(" sign", "Int8",
UInt64(1)));
    columns->columns-
>children.emplace back(create materialized column declaration(" version",
"UInt64", UInt64(1)));
    if (ASTPtr partition expression = getPartitionPolicy(primary keys))
        storage->set(storage->partition_by, partition_expression);
    if (ASTPtr order by expression = getOrderByPolicy(primary keys,
unique_keys, keys, increment_columns))
        storage->set(storage->order_by, order_by_expression);
    storage->set(storage->engine, makeASTFunction("ReplacingMergeTree",
std::make_shared<ASTIdentifier>(version_column_name)));
```

```
CREATE TABLE test.test_table (`primary_key` int PRIMAARY KEY, value
varchar(20)) ENGINE = INNODB;
```





```
CREATE TABLE test.test_table (`primary_key` int, value varchar(20), _sign
Int8 DEFAULT 1, _version UInt64 DEFAULT 1) ENGINE =
ReplacingMergeTree(_version) PARTITION BY intDiv(`primary_key`, 4294967) ORDER
BY (`primary_key`);
```







Check MySQL status

Select history data









```
client.connect();
  client.startBinlogDumpGTID(randomNumber(), mysql_database_name,
metadata.executed_gtid_set);

Buffers buffers(database_name);
  while (!isCancelled())
  {
     BinlogEventPtr binlog_event =
     client.readOneBinlogEvent(std::max(UInt64(1), max_flush_time -
     watch.elapsedMilliseconds()));
     {
        if (binlog_event)
            onEvent(buffers, binlog_event, *metadata);
        if (!buffers.data.empty())
            flushBuffersData(buffers, *metadata);
     }
}
```





```
client.connect();
  client.startBinlogDumpGTID(randomNumber(), mysql_database_name,
metadata.executed_gtid_set);

Buffers buffers(database_name);
  while (!isCancelled())
  {
     BinlogEventPtr binlog_event =
     client.readOneBinlogEvent(std::max(UInt64(1), max_flush_time -
     watch.elapsedMilliseconds()));
     {
        if (binlog_event)
            onEvent(buffers, binlog_event, *metadata);
        if (!buffers.data.empty())
            flushBuffersData(buffers, *metadata);
     }
}
```





```
void MaterializeMySQLSyncThread::onEvent(Buffers & buffers, const
BinlogEventPtr & receive_event, MaterializeMetadata & metadata)
   if (receive event->type() == MYSQL WRITE ROWS EVENT)
   else if (receive_event->type() == MYSQL_DELETE_ROWS_EVENT)
   else if (receive_event->type() == MYSQL_UPDATE_ROWS_EVENT)
   else if (receive_event->type() == MYSQL_QUERY_EVENT)
```





```
void MaterializeMySQLSyncThread::onEvent(Buffers & buffers, const
BinlogEventPtr & receive_event, MaterializeMetadata & metadata)
{
    if (receive_event->type() == MYSQL_WRITE_ROWS_EVENT)
    {
        WriteRowsEvent & write_rows_event = static_cast<WriteRowsEvent
&>(*receive_event);
        Buffers::BufferAndSortingColumnsPtr buffer =
buffers.getTableDataBuffer(write_rows_event.table, global_context);

        size_t bytes = onWriteOrDeleteData<1>(write_rows_event.rows, buffer->first, ++metadata.data_version);

        buffers.add(buffer->first.rows(), buffer->first.bytes(),
write_rows_event.rows.size(), bytes);
    }
    else if (receive_event->type() == MYSQL_DELETE_ROWS_EVENT)
    ...
}
```



```
void MaterializeMySQLSyncThread::onEvent(Buffers & buffers, const
BinlogEventPtr & receive_event, MaterializeMetadata & metadata)
{
    if (receive_event->type() == MYSQL_WRITE_ROWS_EVENT)
        ...
    else if (receive_event->type() == MYSQL_DELETE_ROWS_EVENT)
    {
        DeleteRowsEvent & delete_rows_event = static_cast<DeleteRowsEvent
&>(*receive_event);
        Buffers::BufferAndSortingColumnsPtr buffer =
buffers.getTableDataBuffer(delete_rows_event.table, global_context);
        size_t bytes = onWriteOrDeleteData<-1>(delete_rows_event.rows, buffer->first, ++metadata.data_version);
        buffers.add(buffer->first.rows(), buffer->first.bytes(),
delete_rows_event.rows.size(), bytes);
    }
    else if (receive_event->type() == MYSQL_UPDATE_ROWS_EVENT)
    ...
}
```



```
else if (receive_event->type() == MYSQL_UPDATE_ROWS_EVENT)
        std::vector<bool> writeable rows mask(rows data.size());
        for (size t index = 0; index < rows data.size(); index += 2)</pre>
            writeable rows mask[index + 1] = true;
            writeable_rows_mask[index] = differenceSortingKeys(DB::get<const</pre>
Tuple &>(rows_data[index]), DB::get<const Tuple &>(rows_data[index + 1]),
sorting_columns index);
        for (size t index = 0; index < rows data.size(); index += 2)</pre>
            if (likely(!writeable_rows_mask[index]))
                sign_column_data.emplace_back(1);
                version column data.emplace back(version);
            }
            else
                sign column data.emplace back(-1);
                sign column data.emplace back(1);
                version_column_data.emplace_back(version);
                version column data.emplace back(version);
```





```
void MaterializeMySQLSyncThread::onEvent(Buffers & buffers, const
BinlogEventPtr & receive_event, MaterializeMetadata & metadata)
{
    if (receive_event->type() == MYSQL_WRITE_ROWS_EVENT)
        ...
    else if (receive_event->type() == MYSQL_QUERY_EVENT)
    {
        if (query->as<ASTDropQuery>())
            ...
        else if (query->as<ASTRenameQuery>())
            ...
        else if (query->as<MySQLParser::ASTAlterQuery>())
            ...
        else if (query->as<MySQLParser::ASTCreateQuery>())
            ...
        }
}
```





```
client.connect();
  client.startBinlogDumpGTID(randomNumber(), mysql_database_name,
metadata.executed_gtid_set);

Buffers buffers(database_name);
  while (!isCancelled())
  {
     BinlogEventPtr binlog_event =
     client.readOneBinlogEvent(std::max(UInt64(1), max_flush_time -
     watch.elapsedMilliseconds()));
     {
        if (binlog_event)
            onEvent(buffers, binlog_event, *metadata);
        if (!buffers.data.empty())
            flushBuffersData(buffers, *metadata);
     }
}
```





```
void MaterializeMySQLSyncThread::flushBuffersData(Buffers & buffers,
MaterializeMetadata & metadata)
{
    metadata.transaction(client.getPosition(), [&]()
    { buffers.commit(global_context); });

    const auto & position_message = [&]()
    {
        std::stringstream ss;
        client.getPosition().dump(ss);
        return ss.str();
    };

    LOG_INFO(log, "MySQL executed position: \n {}", position_message());
}
```



F.A.Q

- MySQL 5.6 is not supported
- --binlog-checksum=NONE is not supported

