Temporary Table Design

Kangli Mao Ming Zhang

This document is open to the community

Introduction

A temporary table is a special table whose rows data are only temporarily available.

TiDB will implement both local temporary tables and global temporary tables. The local temporary table is basically compatible with MySQL temporary tables. The global temporary table is a subset of the SQL standard, which supports only one table commit action.

For a local temporary table, it is visible only within the current session, and the table is dropped automatically when the session is closed.

For a global temporary table, the table schema is visible to all the sessions, but the changes into the table are only available within the current transaction, when the transaction commits, all changes to the global temporary table are discarded.

Syntax

The syntax of creating and dropping a global temporary table:
```sql
CREATE GLOBAL TEMPORARY TABLE tbl_name (create_definition)
```

```
        [ENGINE=engine_name]
        ON COMMIT DELETE ROWS;

DROP TABLE tbl_name;
```

The syntax of creating and dropping a session(local) temporary table:
```sql
CREATE TEMPORARY TABLE tbl_name (create_definition)
        [ENGINE=engine_name];

DROP [TEMPORARY] TABLE tbl_name;
```

In the following sections, session temporary tables and local temporary tables are used interchangeably.

Visibility of table definition

There are 2 kinds of table definition visibility.

- Global: global temporary tables are visible to all sessions. These tables only need to be created once and the metadata is persistent.
- Local: local temporary tables are visible to the session that creates them. These tables must be created in the session before being used and the metadata is only kept in memory.

Visibility of data

There are 2 kinds of data visibility:

- Session: when the table is a session temporary table, the data will be kept after a transaction commits. Subsequent transactions will see the data committed by the previous transactions.
- Transaction: when the table is defined as `ON COMMIT DELETE ROWS`, this data will be cleared automatically after a transaction commits. Subsequent transactions won't see the data committed by the previous transactions.

Storage engines

TiDB uses TiKV and TiFlash to store the data of normal tables. This is also true for temporary tables by default because MySQL does so. However, temporary tables sometimes are used to boost performance, so it's also reasonable to support in-memory temporary tables. There are 2 kinds of storage engines available:

- Memory or TempTable: TiDB keeps data in memory, and if the memory consumed by a temporary table exceeds a threshold, the data will be spilled to the local disk on TiDB. This is the default storage engine, which is different from MySQL.
- InnoDB: TiDB stores data on TiKV and TiFlash with more than one replica, just like normal tables. According to MySQL, this is the default storage engine, even for temporary tables.

Motivation or Background

Temporary tables are useful in applications where a result set is to be buffered (temporarily persisted), perhaps because it is constructed by running multiple DML operations.

The purposes of temporary tables include:

- Usability. Temporary tables are typically for temporary use. Applications don't need to frequently truncate the table.
- Performance. In-memory temporary tables are stored in memory, which boosts performance.
- Materialize middle data for queries. Some queries apply internal temporary tables for materializing middle data, such as CTE.

Detailed Design

Metadata

The **table ID** of global temporary tables must be globally unique, while the local temporary tables don't. However, logical or physical plans involve table IDs, which means the temporary table IDs must be different from the normal table IDs. To achieve this goal, it's straightforward to also allocate local temporary table IDs globally.

For a global temporary table, its **table name** should not be duplicated with a normal table. For a local temporary table, when its name conflicts with an existing table, it will take a higher priority. We can keep the temporary tables in a local schema, and overwrite the original one. The databases where temporary tables belong depend on the identifier in `CREATE TABLE` statements, just like normal tables.

Since the metadata of global temporary tables are persistent on TiKV, it's straightforward to execute DDL in the same procedure as normal tables. However, the metadata of local temporary tables are only kept in the memory of the current TiDB instance, so we can bypass the complex **online DDL procedure**. We need only to generate the metadata locally and then merge it into the information schema. Thus, users cannot see the DDL jobs of local temporary tables through `ADMIN SHOW DDL JOBS`.

Local temporary tables don't support **altering table** operations because few users will do that. TiDB should report errors when users try to do that.

As all DDL statements do, any DDL on a global temporary table will cause an **implicit commit**. However, creating and dropping a local temporary table doesn't cause an implicit commit, according to [the MySQL documentation](https://dev.mysql.com/doc/refman/8.0/en/implicit-commit.html).

Each temporary table belongs to its own database. Local temporary tables have a very loose relationship with databases. Dropping a database does not automatically drop any local temporary tables created within that database. The local temporary tables still stay in a virtual database with the same name.

Truncating global temporary tables also conforms to the online DDL procedure, which affects other sessions. However, it's different for local temporary tables because the metadata is kept in memory. Truncating local temporary tables just drops the current metadata and creates a new one in memory.

DDL operations, including those using `INSTANT` and `COPY` algorithms, are simple to apply on global temporary tables. For example, **adding an index** on a global temporary table is easy, because the table must be empty before adding the index. This benefits from implicit commit before adding the index. Local temporary tables, on the other hand, do support adding indexes, as other altering table operations.

Some options in `CREATE TABLE` statements are not suitable for temporary tables. These options include: `**AUTO_RANDOM**`, `**SHARD_ROW_ID_BITS**`, `**PRE_SPLIT_REGIONS**`, `**PARTITION BY**`, `**FOREIGN KEY**`. Similarly, some related DDL operations are not supported, such as `**SPLIT TABLE**`, `**SHOW TABLE REGIONS**`, `**ALTER PLACEMENT POLICY**`. Table partition option is useless to a temporary table in the real use cases, so it's also not supported. Errors should be reported when users declare such options in the statements.

Since some options are not suitable for temporary tables, when a user creates a temporary table from the `**CREATE TABLE LIKE**` statement and the source table has these options, an error should be reported.

Other options will be kept. For example, clustered indexes and secondary indexes are kept because they can improve performance.

Altering table types is not allowed, since few users will do that:

- Altering a temporary table to a normal table or conversely.
- Altering a global temporary table to a local temporary table or conversely.
- Altering the storage engine of temporary tables.

The result of `**SHOW TABLES**` contains global temporary tables, but not local temporary tables. The motivation is that local temporary tables are only for temporary usage, so the

user should know what he is doing. However, `**SHOW CREATE TABLE**` works for all temporary tables.

Similarly, system tables `TABLES` and `COLUMNS` in `**information_schema**` do not contain local temporary tables, but they contain global temporary tables. For global temporary tables, the value of field `TABLE_TYPE` in `information_schema.TABLES` is `GLOBAL TEMPORARY TABLE`.

`**ADMIN CHECK TABLE**` and `**ADMIN CHECKSUM TABLE**` are used to check data integrity of the table. Data of temporary tables might also be corrupted due to unexpected problems, but it's impossible to check them because they are invisible to other sessions. So TiDB doesn't support these commands.

Creating **views** on a global temporary table makes sense, and the view will also be persistent on TiKV. However, it's unreasonable to persist a view that is based on a local temporary table, because the table will be discarded when the session ends.

As the metadata of global temporary tables are persistent on TiKV, the **binlog** of DDL should also be exported. However, this is unnecessary for local temporary tables.

Optimizer

Statistics of temporary tables are very different from those of normal tables. The data of each temporary table is invisible to other sessions, so it's unnecessary to persist statistics on TiKV. On the other hand, even if the sizes of temporary tables are relatively small, it's also necessary to consider statistics since it may improve the query performance significantly.

Updating statistics is a little different from normal tables. We can't rely on `**AUTO ANALYZE**` anymore, because the lifecycle of one session is relatively short, it's unlikely to wait for `AUTO ANALYZE`. What's more, `AUTO ANALYZE` runs in background sessions, which means they can't visit the temporary data.

It's also unreasonable to force users to run `**ANALYZE TABLE**` periodically in applications. Intuitively, there are 2 ways to maintain statistics:

- Update statistics once updating the table. When users run `ANALYZE TABLE`, TiDB updates statistics in the current session, instead of in background sessions.
- Instead of maintaining statistics, collecting needed statistics before each query is another option. The collection can be boosted by sampling. `ANALYZE TABLE` needs to do nothing in this way.

Both ways are also easy to implement, and it's not concrete that we really need statistics for temporary tables for now. So we only maintain row count, and just skip maintaining NDV, selectivity and others until there's a need.

Statistics of the same global temporary table are different in different sessions, so every session keeps a copy of statistics for each global temporary table. However, the statistics of normal tables are stored in a global cache, which can be visited concurrently. So there needs to be a way to store the statistics of global temporary tables separately.

Obviously, the **cost** of reading temporary tables is much lower than reading normal tables, since TiDB doesn't need to request TiKV or consume any network resources. So the factors of such operations should be lower or 0.

SQL binding is used to bind statements with specific execution plans. Global SQL binding affects all sessions and session SQL binding affects the current session. Since local temporary tables are invisible to other sessions, global SQL binding is meaningless. TiDB should report errors when users try to use global SQL binding on local temporary tables. Creating global or session SQL binding on global temporary tables, and creating session SQL binding on local temporary tables are both allowed.

Baseline is used to capture the plans of statements that appear more than once. The appearance of statements is counted by SQL digests. Even if all sessions share the same global temporary table definitions, the data and statistics is different from one session to another. Thus baseline and SPM is useless for temporary tables. TiDB will ignore this feature for temporary tables.

Prepared plan cache is used to cache plans for prepared statements to avoid duplicate optimization. Each session has a cache and the scope of each cache is the current session. Even if the cached plan stays in the cache after the temporary table is dropped, the plan won't take effect and will be removed by the LRU list finally. So we just leave it as it was.

Storage

Before going further to the executor, we need to determine the storage form of temporary tables.

Basically, there are a few alternatives to store the data of in-memory temporary tables, and the most promising ones are:

- Unistore. Unistore is a module that simulates TiKV on a standalone TiDB instance.
- UnionScan. UnionScan is a module that unites membuffer and TiKV data. Membuffer buffers the dirty data a transaction writes. Query operators read UnionScan and UnionScan will read both buffered data and persistent data. Thus, if the persistent part is always empty, then the UnionScan itself is a temporary table.

	Unistore	UnionScan
execution	Y	Y
indexes	Y	Y

spilling to disk	Y	N
MVCC	Y	N

TiDB uses UnionScan to store the data of temporary tables for the following reasons:

- **The performance should be better.** It can cut down the marshal/unmarshal cost and a lot of the coprocessor code path, which is inevitable in the Unistore as a standalone storage engine.
- **The implementation is easier.** As long as we don't take the spilling to disk feature into consideration for now, the global temporary table is almost handly. And we do not bother by the tons of background goroutines of the Unistore engine when dealing with resource releasing. How to keep the atomicity is another headache if we choose the Unistore, imaging that a transaction would write to both temporary tables and normal tables at the same time.
- **A lower risk of introducing bugs.** Although we implement the coprocessor in the Unistore, we just use it for the testing purpose, and we also have some experimental features first implemented in the Unistore, so its behavior may slightly differ from the real TiKV, and that difference would introduce bugs.

Nothing needs to be changed for the KV encoding, the temporary table uses the same strategy with the normal table.

When the memory consumed by a temporary table exceeds a threshold, the data should be spilled to the local disk to avoid OOM. The threshold is defined by the system variable `temptable_max_ram`, which is 1G by default. Membuffer does not support disk storage for now, so we need to implement it.

A possible implementation is to use the AOF (append-only file) persistent storage, the membuffer is implemented as a red-black tree and its node is allocated from a customized allocator. That allocator manages the memory in an arena manner. A continuous block of memory becomes an arena block and an arena consists of several arena blocks. We can dump those blocks into the disk and maintain some LRU records for them. A precondition of the AOF implementation is that once a block is dumped, it is never modified. Currently, the value of a node is append-only, but the key is modified in-place, so some changes are necessary. We can keep all the red-black tree nodes in memory, while keeping part of the key-value data in the disk.

Another option is changing the red-black tree to a B+ tree, this option is more disk friendly but the change takes more effort.

When a temporary table needs to be **cleared**, its disk data should also be cleared. That can be done asynchronously by a background goroutine. The goroutine needs to know whether a file is in use. So TiDB needs to maintain a map which contains in-use files, each session updates the map periodically if it still needs the file. If some files are not touched for a long time, we can treat the session as crashed and collect the disk files.

For on-disk temporary tables, it's straightforward to store them on TiKV like normal tables. Since each local temporary table has a unique table ID, the data from different sessions are separate. However, multiple sessions using the same global temporary table will share the same table ID, which means the data from multiple sessions is continuous and stored together. Sessions will affect each other in this case.

Clearing the data of on-disk temporary tables is a little different from in-memory temporary tables. When a TiDB instance is down, the storage space needs to be collected by other TiDB instances. Thus the maintenance of in-use temporary tables should be on the TiKV side.

Executor

For normal tables, **auto-increment IDs** and row IDs are allocated in batches from TiKV, which significantly improves performance. However, as temporary tables are usually inserted, it may cause write hotspots on TiKV. So the best way is to allocate IDs locally on TiDB. The ID of a global temporary table is allocated separately among sessions and rebases to 0 every time a transaction ends. That means, each session needs a copy of allocators, rather than sharing the same allocators.

Besides, `last_insert_id` is also affected by inserting into temporary tables.

Since the data of in-memory temporary tables are not needed to be cached anymore, **coprocessor cache** and **point-get cache** are ignored. But they still work for on-disk temporary tables.

Follower read indicates TiDB to read the follower replicas to release the load of leaders. For in-memory temporary tables, this hint is ignored. But it still works for on-disk temporary tables.

Users can also choose the storage engine to read by setting **`tidb_isolation_read_engines`**. For in-memory temporary tables, this setting will also be ignored. But it still works for on-disk temporary tables.

Since in-memory temporary tables are not persistent on TiKV or TiFlash, writing **binlog** for DML is also unnecessary. This also stays true for on-disk temporary tables, because data is invisible to other sessions.

It's straightforward to support **MPP** on on-disk temporary tables, because the data is also synchronized to TiFlash. Most operators on in-memory temporary tables can still be processed in TiDB, such as Aggregation and TopN. These operators will not cost much memory because the sizes of in-memory temporary tables are relatively small.

However, joining normal tables with in-memory temporary tables might be a problem, because the sizes of normal tables might be huge and thus merge sort join will cause

OOM, while hash join and index lookup join will be too slow. Supporting broadcast join and shuffled hash join on in-memory temporary tables is very difficult. Fortunately, MPP typically happens in OLAP applications, where writing and scanning duration is relatively short compared to computing duration. So users can choose to define on-disk temporary tables in this case.

Transaction

Because it's rare to read historical data from a temporary table, temporary tables don't support features that rely on **MVCC**, like flashback tables, recover tables, stale reads, and historical reads. Errors will be reported when users execute such statements on temporary tables.

If a transaction only writes to temporary tables without normal tables, it does not really need a **TSO** for committing, because MVCC is unsupported and the data is invisible to other sessions. But for the sake of simplicity, we still fetch commitTS just like normal tables.

When a transaction **commits**, the data in in-memory temporary tables should not be committed on TiKV. When a transaction rollbacks, operations on normal tables and temporary tables should be rolled back together. The data in on-disk temporary tables will be committed. However, this can be omitted by global on-disk temporary tables, because the data will be cleared anyway. TiDB can iterate KV ranges to filter out the data.

Since there won't be concurrent modifications on the same temporary table, there won't be lock conflicts. So `**FOR UPDATE**` and `LOCK IN SHARE MODE` clauses will be ignored.

Transactions might **retry** write operations when commit fails. DML on normal tables might rely on the data on temporary tables, so DML on temporary tables should also be retried. For example:

- `INSERT INTO normal_table SELECT * FROM temp_table`
- `UPDATE normal_table, temp_table SET … WHERE normal_table.id=temp_table.id`

If DML on temporary tables is not retried, such statements won't write any data.

Specially, as mentioned above, creating and dropping local temporary tables might also be in a transaction, but they needn't be retried.

TiDB comes with an optimization when the variable `**tidb_constraint_check_in_place**` is disabled: checking for duplicate values in UNIQUE indexes is deferred until the transaction commits. For those cases where temporary tables skip 2PC, this optimization should be disabled.

Local transactions are special transactions that fetch TSO from the local PD. They can not access the data that is bound to the current available zone. Although temporary tables are

not bound to any zones, they are invisible to other sessions, which means local transactions can still guarantee linearizability even when they access temporary tables.

Schema change on a global temporary table may happen during a transaction which writes to the temporary table. Unlike normal tables, the transaction won't overwrite other transactions, so it's fine to commit. Schema change on a local temporary table will never happen during a transaction which writes to the temporary table.

Privileges

Creating a local temporary table checks the `CREATE TEMPORARY TABLES` privilege. No access rights are checked when dropping a local temporary table, according to [the MySQL documentation](https://dev.mysql.com/doc/refman/8.0/en/drop-table.html).

All DDL on global temporary tables check the corresponding privileges like normal tables do.

Writing to a global temporary table checks the privileges like the normal table. But there is no privilege check for a local temporary table.

Privileges can not be granted to local temporary tables, because the tables are invisible to other users. Granting privileges to global temporary tables is possible.

Ecosystem Tools

As mentioned above, DDL binlog of global temporary tables needs to be recorded, but not for local temporary tables. DML binlog is always skipped for temporary tables. DDL of global temporary tables should be supported by all data migration tools whose upstream is TiDB, such as Dumpling, TiDB-binlog, and TiCDC.

Since `information_schema.tables` lists global temporary tables, these tables will be processed by tools like **Dumpling**. Fortunately, querying global temporary tables in a new session just returns empty results, so nothing needs to be handled.

When backup tools read TiKV data, the data of temporary tables should never be read. However, on-disk temporary tables are stored on TiKV and TiFlash, so they need to be ignored by those tools, such as **BR** and **TiCDC**. Since these tools can see the metadata, they should also be capable of skipping tables that are not normal tables.

Telemetry is used to report the usage information of various features in TiDB. Global and local temporary tables will be reported by telemetry separately, because the scenarios of them are different.

Internal temporary tables

In MySQL, temporary tables will be used internally as infrastructures of other features, such as CTE, derived tables, and UNION statements. These temporary tables are called [internal temporary tables](https://dev.mysql.com/doc/refman/8.0/en/internal-temporary-tables.html).

In the future, temporary tables will probably be used internally for some new features, such as CTE. So TiDB should be capable of creating a local temporary table when it's given a list of columns. Internal temporary tables might be in-memory or on-disk, depending on the optimizer. Physical plans that don't apply MPP should use in-memory temporary tables, otherwise they will use on-disk temporary tables.

Transactions that apply internal temporary tables might be read-only, but there are some steps which write to TiKV:

- Assigning table ID for temporary tables
- Writing to on-disk temporary tables

Writing to on-disk temporary tables is inevitable, so the best way is NOT to report errors in this case.

When executing a CTE in MPP mode, TiFlash **has to write to the temporary table by itself**, because the middle data is generated by TiFlash.

Modification to the write path

The data of the temporary table is all in the membuffer of UnionScan. Writing to the temporary table is writing to the transaction cache.

The data writing path in TiDB is typically through the `table.Table` interface, rows data are converted into key-values, and then written to TiKV via 2PC. For normal transactions, the data is cached in the membuffer until the transaction commits. The temporary table should never be written to the real TiKV.

A transaction can write to both the temporary table and the normal table. Should the temporary table share the membuffer with the normal table, or use a separate one?
It's better to share the membuffer so the commit/rollback operation can be atomic. The risk is that in case of a bug, the temporary table data may be written to the real table.

For global temporary tables, since the transaction commits automatically clears the temporary table, we can filter out and discard the transaction cache data from the temporary table when committing, and the implementation is relatively simple.

For local temporary tables, the temporary table data is cleared at the end of the session, so the data should survive the transaction's commit. We can copy the key-value data belonging to the temporary table to another place and use it in the later read operation.

Modification to the read path

In TiDB, reading is implemented by the coprocessor. Read operations should see their own writes. TiDB uses a UnionScan operator on top of the coprocessor's executor. This operator takes the data read by the coprocessor as a snapshot, then merges it with the membuffer, and passes the result to its parent operator.

For transactional temporary tables, there is no need to do any additional changes, the current code works well.

For the local temporary table, the key-value data of the temporary table in the current transaction should be taken out. We keep a temporary table membuffer in the session variable and if it is not null, the UnionScan operator needs to combine it with the original data. So now the data hierarchy looks like this:

> TiKV snapshot => Old membuffer => current transaction’s membuffer

Upgrade and Downgrade Compatibility

When users downgrade TiDB to an older version after they have created a global temporary table, the table should not be seen by the older version. Otherwise, they might write to the table and then upgrade TiDB, which will be a problem.

Test Design

A brief description of how the implementation will be tested. Both the integration test and the unit test should be considered.

Functional Tests

It’s used to ensure the basic feature function works as expected. Both the integration test and the unit test should be considered.

- DDL
    - Create table / create table like
    - Drop table
    - Truncate table
    - Other DDL (for global)
- DML
    - Insert / replace / update / delete / load data
    - All kinds of query operators
    - Show create table / show tables
- Transaction

    - Atomicity (for local)
    - Isolation (RC / SI, linearizability)
    - Data visibility between transactions
    - Optimistic / Pessimistic transactions
- Information_schema
    - Tables
    - Columns
- Privileges
    - DDL
    - DML

Scenario Tests

It's used to ensure this feature works as expected in some common scenarios.

Before we implement the spilling to disk feature, we have to know the memory usage for some scenarios. For example, 100M for each temporary table, and 500-600 concurrent connections, how much memory does TiDB use.

Compatibility Tests

Feature	Compatibility	Temporary table type	Reason
placement rules	Report error: not support		Meaningless
partition	Report error: not support		Meaningless
show table regions / split table / pre_split_regions	Report error: not support		Meaningless
stale read / historical read	Report error: not support		Meaningless
auto_random / shard_row_id_bits	Report error: not support		No need to release writing hotspots
flashback / recover table	Report error: not support		Meaningless
global SQL binding	Report error: not support	local	Bindings are meaningless after session ends & Tables are different among sessions
view	Report error: not support	local	Views are meaningless after session ends & Tables are

			different among sessions
copr cache	Ignore this setting	in-memory	No need to cache
point get cache	Ignore this setting	in-memory	No need to cache
follower read	Ignore this setting	in-memory	Data is neither on TiKV nor on TiFlash
read engine	Ignore this setting	in-memory	Data is neither on TiKV nor on TiFlash
GC	Ignore this setting		No need to GC data
select for update / lock in share mode	Ignore this setting		No lock is needed
baseline / SPM	Ignore this setting		Tables / stats are different among sessions
tidb_constraint_check_in_place	Ignore this setting	in-memory & local	Data is not committed on TiKV
auto analyze	Ignore this setting		Background sessions can't access private data
global txn / local txn	Need to deal with		No limitation for read / write
analyze	Need to deal with		Update in-memory stats in the current session instead of in system sessions
broadcast join / shuffle hash join	Need to deal with	on-disk	Only support on-disk temporary tables
telemetry	Need to deal with		Report the usage of temporary tables
view	Need to test	global	Drop views on dropping temporary tables
auto_increment / last_insert_id	Need to test		
alter table	Report error: not support	local	Too hard to support and unnecessary
all hints	Need to test		
plan cache	Need to test		plan cache is session-scope
show fields / index / keys	Need to test		
SQL binding	Need to test		

clustered index	Need to test		
async commit / 1PC	Need to test		
checksum / check table	Report error: not support		
collation / charset	Need to test		
batch insert	Need to test		
feedback	Need to test		
statements_summary / slow_query	Need to test		SQL normalization
big transaction	Need to test		
memory tracker	Need to test		
explain / explain analyze	Need to test		

Compatibility with other external components, like TiDB, PD, TiKV, TiFlash, BR, TiCDC, Dumpling, TiUP, K8s, etc.
Upgrade compatibility
Downgrade compatibility

Benchmark Tests

The following two parts need to be measured:
measure the performance of this feature under different parameters
measure the performance influence on the online workload

sysbench for the temporary table, comparing its performance with the normal table. It means comparing the performance of an in-memory system with a distributed system.

Investigation & Alternatives

How do other systems solve this issue? What other designs have been considered and what is the rationale for not choosing them?

MySQL documentation for the temporary table
https://dev.mysql.com/doc/refman/8.0/en/internal-temporary-tables.html

CockroachDB just uses the normal table as the temporary table. All the temporary tables are stored in a special schema, and it is scanned and cleaned periodically. If a session is finished, the temporary tables of that session are also cleaned.

https://github.com/cockroachdb/cockroach/issues/5807

Oracle uses global temporary tables in the old versions, and later on they also have the private temporary tables. The private temporary table in Oracle looks like MySQL, those tables are visible to the session rather than global. For global temporary tables, Oracle does not need to handle the schema change, because `alter table` is not allowed when some transactions are using the table.
https://docs.oracle.com/cd/B28359_01/server.111/b28310/tables003.htm#ADMIN11633

Development Plan

- Stage 1(Support global, in-memory temporary table)
    - The architecture design for global/local, session/transaction, in-memory/disk spilling (L)
    - DDL（2XL)
    - DML (2XL)
    - Backup and recover the temporary table meta information (L)
    - Sync the temporary table DDL through CDC or Binlog to the downstream
    - Privilege (L)
    - Compatibility with other features (2XL)
    - Performance test (L)
- Stage 2 (Support local, in-memory temporary table)
    - create/drop session temporary table (3XL)
    - Compatibility with MySQL 8.0 temporary table (in-memory)
    - Compatibility with other features (2XL)
- Stage 3
    - Support spilling to disk for all kind of temporary tables (4XL)

Design Reviewers

Time	Notes	Recorder	Decision