# A Lightweight and Efficient Temporal Database Management System in TDSQL

Wei Lu[†], Zhanhao Zhao[†], Xiaoyu Wang[§], Haixiang Li[§],
Zhenmiao Zhang[†], Zhiyu Shui[†], Sheng Ye[§], Anqun Pan[§], Xiaoyong Du[†][*]

[†]*School of Information and DEKE, MOE, Renmin University of China, Beijing, China*
[§]*Tencent Inc., China*
{lu-wei, zhanhaozhao, zhzm, shuizhiyu, duyong}@ruc.edu.cn
{blueseali, xiaoyuwang, shengye, aaronpan}@tencent.com

## ABSTRACT

Driven by the recent adoption of temporal expressions into SQL:2011, extensions of temporal support in conventional database management systems (a.b.a. DBMSs) have re-emerged as a research hotspot. In this paper, we present a lightweight yet efficient built-in temporal implementation in Tencent's distributed database management system, namely TDSQL. The novelty of TDSQL's temporal implementation includes: (1) a new temporal data model with the extension of SQL:2011, (2) a built-in temporal implementation with various optimizations, which are also applicable to other DBMSs, and (3) a low-storage-consumption in which only data changes are maintained. For the repeatability purpose, we elaborate the integration of our proposed techniques into MySQL. We conduct extensive experiments on both real-life dataset and synthetic TPC benchmarks by comparing TDSQL with other temporal databases. The results show that TDSQL is lightweight and efficient.

**PVLDB Reference Format:**
Wei Lu, Zhanhao Zhao, Xiaoyu Wang, Haixiang Li, Zhenmiao Zhang, Zhiyu Shui, Sheng Ye, Anqun Pan, Xiaoyong Du. A Lightweight and Efficient Temporal Database Management System in TDSQL. *PVLDB*, 12(12): 2035-2046, 2019.
DOI: https://doi.org/10.14778/3352063.3352122

## 1. INTRODUCTION

The study on temporal data management has been going on for decades, but only recently has some progress been made. Temporal data management by applications brings prohibitively expensive development and maintenance overhead. Instead, extensions to support temporal data management in conventional DBMSs have been extensively explored. Nevertheless, the temporal support offered by commercially available software tools and systems is still quite limited. Until recently, the adoption of temporal expressions into SQL:2011 makes the major DBMSs provide built-in temporal support, typically including: (1) extension of non-temporal tables to temporal tables, (2) a unified management of both current data and historical data in a single database, (3) query rewrite functionality by expressing the semantics of temporal queries in equivalent conventional (non-temporal) SQL statements.

In SQL:2011, the temporal features mainly include temporal table definitions, temporal queries, and others (e.g., temporal constraints). As compared to the non-temporal counterpart, a temporal table associates either valid time, or transaction time, or both. Either valid time or transaction time is a closed-open period (i.e., time interval) $[s, t)$, with $s$ as *start time* and $t$ as *end time*. Valid time is a time period during which a fact was/is/will be true in reality, and transaction time is a time period during which a fact is/was recorded (i.e., current/historical) in the database.

EXAMPLE 1. *Table 1 shows an example of player account balance. TT is the transaction time. As we can see,* $r_{1.4}, r_{2.2}$, $r_{3.1}$ *are the current records of players James, David, and Jack, respectively, while the remaining are historical record sets of James, David, respectively. According to the temporal expressions of SQL:2011, by specifying the qualifier 'FOR TT FROM TIMESTAMP 2018-11-08 00:00:00 TO TIMESTAMP 2018-11-10 23:59:59', both current and historical records* $r_{1.3}, r_{2.2}$, $r_{3.1}$ *are returned.* □

Thus far, we have witnessed a big burst of temporal support in conventional DBMSs, such as Oracle [3], Teradata [6], MariaDB [2], SQL Server [5]. However, they still suffer from limited expressiveness and poor performance.

First, existing temporal data model is inadequate. Due to some unexpectedly erroneous transaction made by operators, logical data corruption, e.g., accidental deletion of valuable data, is difficult to avoid. Based on the existing temporal data model, a temporal database is able to rewind the states of the data to a specified point in time based on

[*]Haixiang Li and Xiaoyong Du are the corresponding authors.

*Proceedings of the VLDB Endowment,* Vol. 12, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3352063.3352122

Table 1: Account balance

| ID | Player | Bal | Transaction Time |
|---|---|---|---|
| $r_{1.1}$ | James | 0 | [2018-05-20 06:20:00,2018-10-21 00:30:00) |
| $r_{1.2}$ | James | 50 | [2018-10-21 00:30:00,2018-11-01 09:01:41) |
| $r_{1.3}$ | James | 1000 | [2018-11-01 09:01:41,2018-11-12 00:00:10) |
| $r_{1.4}$ | James | 2000 | [2018-11-12 00:00:10,$\infty$) |
| $r_{2.1}$ | David | 150 | [2018-10-20 20:10:10,2018-10-20 20:40:00) |
| $r_{2.2}$ | David | 200 | [2018-10-20 20:40:00,$\infty$) |
| $r_{3.1}$ | Jack | 200 | [2018-11-08 10:12:43,$\infty$) |

the transaction time of each record. Nevertheless, it cannot figure out the records in an erroneous transaction simply based on the transaction time in that many transactions could occur in the same transaction time. Thus, the recovery of logical data corruption for a given erroneous transaction is hardly to accomplish.

Second, the transaction time in the temporal data model is difficult to set/update. Consider the majority of conventional DBMSs, including Oracle, SQL Server, MySQL, are MVCC-based. Theoretically, for a given record $r$, the start time (or end time) in its transaction time should be set to the commit time of the transaction that creates (or updates/deletes) $r$ based on the snapshot isolation theorem [7, 23, 26]. Nevertheless, most of the existing temporal implementations just pick up the time when transaction starts to execute, or the time of the operation that inserts/updates/deletes the record (note that conventional DBMSs do not maintain the commit time of the transaction with each record by taking into consideration the performance of the whole system) just because SQL:2011 does not enforce to use the commit time [21]. We argue that these implementations could potentially cause an incorrect result based on the snapshot isolation.

Third, temporal query processing suffers from poor performance. On one hand, by introducing the temporal features, the performance of conventional DBMSs degrades significantly (see our experimental section). The reason is that existing temporal implementations store history and current data separately, i.e., historical table and current table, to skip over historical data for current data query processing, which is deemed to the dominant temporal queries. However, this separation could degrade the throughput of conventional transactional workloads since any update/delete will cause two tables to be concurrently updated. On the other hand, temporal data is maintained in an append-only mode, causing an ever-increasing size of temporal data. The overhead of maintaining large volume of temporal data degrades the query performance.

To address the above issues, in this paper, we propose a lightweight yet efficient built-in temporal implementation in TDSQL. We make the following contributions.

• We present a new temporal data model. As compared to the non-temporal counterpart, a temporal relation under the new model can have two transaction IDs, besides the valid/-transaction time period, defined in SQL:2011. One ID corresponds to the transaction that creates the record, and the other ID corresponds the transaction that deletes/updates the record. By introducing the transaction IDs, it is able to identify all records that are inserted/updated/deleted in the same transaction, thus achieving the recovery of logical data corruption. More importantly, temporal join queries taking the transaction time as the join key are able to be enhanced by taking the transaction ID as the join key instead.

• We propose a built-in temporal implementation with various optimizations in TDSQL, which is also applicable to other DBMSs.

First, our implementations of temporal data storage management are almost non-invasive. Like other temporal implementations in the conventional DBMSs, we use a historical table and a current table to store historical and current data separately. In existing implementations, any update/delete of a current record will result in a synchronous migration of newly generated historical records to the historical table. On the contrary, we propose an asynchronous data migration strategy, i.e., upon any update/delete of a current record will not cause an immediate data migration. Instead, all newly generated historical records are migrated to the historical table only when the database system starts to reclaim the storage occupied by records that are deleted or obsoleted, which is also known as VACUUM in PostgreSQL and PURGE in MySQL. This late data migration transfers historical data in batch and is non-invasive to the originally transactional systems. Further, we propose a key/-value store based approach to efficiently managing the historical data by maintaining data changes only, thus reducing the size of storage space.

Second, in response to the challenge that the transaction time is difficult to set and update, we build an efficient transaction status manager. It maintains the status for the transactions, including the transaction commit time, in a transaction log. A special design of the manager makes the retrieval of the commit time of a given transaction ID at most one I/O cost. During the data migration, we update their transaction time for each of newly generated historical record based on its transaction ID. For the current records and historical records that have not yet been transferred to the historical table, we are still able to obtain the transaction time based on their transaction IDs via the manager.

Third, to support temporal query processing, we extend parser, query executor, storage engine of TDSQL. TDSQL's temporal implementation supports all temporal features defined in SQL:2011. For valid-time qualifiers in the temporal query, we transform the temporal operations into equivalent non-temporal operations; while for transaction-time qualifiers, we provide a native operator to retrieve current and historical data with various optimizations.

• We conduct extensive experiments on both real and synthetic TPC benchmarks by comparing TDSQL with Oracle, SQL Server, and MariaDB. The results show that TDSQL almost has the minimal performance loss (only 7% on average) by introducing the temporal features, and performs the best for most of the temporal queries.

The rest of the paper is organized below. Section 2 discusses related work. Section 3 formalizes our new temporal data model. Section 4 outlines the system architecture. Section 4 elaborates temporal query processing and storage management. Section 6 presents the implementation. Section 7 we present three real temporal applications in Tencent. Section 8 reports the experimental results and Section 9 concludes the paper.

## 2. RELATED WORK

The study on temporal data management has been going on for decades, mainly in the fields of data model development, query processing, and implementations.

Early work until 1990s mainly focused on the consensus glossary of concepts for data modeling [12, 13, 14, 15, 16]. At this stage, the contributions mainly include temporal table definitions, temporal constraints and temporal queries. As compared to the non-temporal counterpart, a temporal relation associates time, which is multi-dimensional, and can be either valid time or transaction time, or other type of time. The semantics of integrity constraints in the temporal data model is also enriched [10, 27]. Entity integrity does not enforce the uniqueness of the primary key. Instead, it requires that no intersection exists between valid times of any two records with the same primary key; while for reference integrity, there must exist one matching record in the parent table whose valid time contains the valid time of the child record. As compared to the regular query syntax, temporal queries are formulated by expressing filtering conditions as period predicates with extensive research work on this field [8, 21].

Extensive efforts have been devoted to build the temporal implementation outside or inside conventional DBMSs. After attempts with many years to build the implementation on top of conventional DBMSs, such as Oracle [3], DB2 [1], and Ingres [28], it is well recognized that the cost, brought by the development and maintenance of application programs, is prohibitively expensive. For this reason, since the late 1990s, extensions to support temporal data management using SQL have been extensively explored [9, 17, 25, 31]. Although a set of temporal extensions, like TSQL2 [11], were submitted for standardization, these attempts are not successful until the adoption of SQL:2011 [21]. In response to SQL:2011, the mainstream of both commercial and open-source database management systems, including Oracle [3], IBM DB2 [1], Teradata [6], PostgreSQL [4], have been dedicated to offer SQL extensions for managing temporal data based on the newly standardized temporal features. Oracle introduces the Automatic Undo Management (AUM) system to manage historical data and answers temporal queries through views executing on both historical and current data. SQL Server, DB2, and Teradata utilize current and history tables to store current and historical data separately. Users issue queries over the current table and the system will retrieve the history table as needed based on the specified point in time or between two specified points in time. ImmortalDB [24] is an well-design research prototype that provides transaction time support built into the SQL Server, but it may suffer from massive storage consumption since only the straightforward storage strategy has been employed. Orthogonal to the temporal implementation, a large number of temporal data access methods are proposed [18, 19, 22, 29, 30]. Many of them are either B+-trees, or R-trees which are widely used in conventional DBMSs. These access methods are potentially used to speed up the query performance. We do not list many other access methods which are not variants of B+-trees or R-trees in that integrating them to DBMSs is not trivial.

## 3. TEMPORAL FEATURES OF TDSQL

In this section, we describe the temporal features of our system in terms of data model, temporal queries, and data constraints.

### 3.1 Temporal Data Model

We support either of the following data models.

• **Valid-time data model.** As compared to the non-temporal counterpart, a relation $R$ under this model associates valid time. Let $\{U, VT\}$ be the attributes of $R$, where $U$ is the attribute set of the non-temporal counterpart for $R$, and $VT$ is the valid-time period.

• **Transaction-time data model.** As compared to the non-temporal counterpart, a relation $R$ under this model associates transaction time and transaction IDs. We denote $\{U, TT, CID, UID\}$ as the attributes of $R$, where $U$ is the attribute set of the non-temporal counterpart for $R$, and $TT$ is the transaction-time period, $CID$ is the transaction ID that creates the record, and $UID$ is the transaction ID that updates/deletes the record.

• **Bi-temporal data model.** A relation $R$ associates both valid-time, transaction-time and transaction IDs, i.e., $R$ has attributes $\{U, VT, TT, CID, UID\}$.

We refer to a relation as a valid-time relation if it merely has valid time, and we make similar definitions for transaction-time relation and bi-temporal relation. We denote $VT$ as a closed-open period $[VT.st, VT.ed)$, and $TT$ as $[TT.st, TT.ed)$. $VT.st, VT.ed, TT.st, TT.ed$ are four time instants. For valid-time, a record is either *currently valid* if $VT.st \leq$ current time $< VT.ed$, or *historical valid* if $VT.ed \leq$ current time, or *future valid* if $VT.st >$ current time. For transaction time, a record is said to be a *current record* if $TT.st \leq$ current time $< TT.ed$, and a *historical record* if $TT.ed \leq$ current time.

### 3.2 Temporal Syntax

We introduce the temporal syntax of TDSQL below.

#### 3.2.1 *Creating Temporal Tables*

As compared to the non-temporal counterpart, a valid-time relation is defined using the following SQL statement:

```
CREATE TABLE R (ID INTEGER, Period VT)
```

A transaction-time relation is created by adding a schema-level qualifier 'WITH SYSTEM VERSIONING':

```
CREATE TABLE R (
  ID INTEGER
) WITH SYSTEM VERSIONING
```

The syntax of creating a bi-temporal relation is the combination of above.

#### 3.2.2 *Valid-time Queries*

A valid-time query is defined as queries on valid-time relations. As compared to the regular SQL syntax, a valid-time query can add valid-time predicates, like OVERLAPS and CONTAINS, as the period predicates, which work together with other regular predicates in the WHERE conditions.

#### 3.2.3 *Transaction-time Queries*

A transaction-time query is defined as queries over transaction relations. As compared to the regular SQL syntax, it is syntactic extended in terms of the transaction time: (1) FOR TT AS OF *t*, which restricts records that are readable at *t*, (2) FOR TT FROM *t1* TO *t2*, which restricts records that are readable from *t1* to (but not include) *t2*, (3) FOR TT BETWEEN *t1* AND *t2*, which restricts records that are readable from *t1* to (and include) *t2*.

### *3.2.4 Transaction ID Queries*

New temporal queries are enriched by introducing transaction IDs. On one hand, the join operation is extended based on the transaction IDs, e.g., a reconciliation requires a join operation on the account balance table ($R$) and the expense statement table ($W$), shown below:

```
SELECT * FROM (
  R FOR TT FROM ts1 TO ts2 as A
  FULL OUTER JOIN
  R FOR TT FROM ts1 TO ts2 as B
  ON A.UID = B.CID
)
FULL OUTER JOIN
W FOR TT FROM ts1 TO ts2 as C
ON B.CID = C.UID
```

On the other hand, logical data corruption recovery (a.b.a. LDCR) is fully supported. A basic LDCR is to rewind the state of table $R$ to a given time $t$ below:

```
REPLACE INTO R
  (SELECT * FROM R FOR TT AS OF t)
```

By introducing transaction IDs, LDCR is extended to support transaction-level recovery via the following syntax:

```
REWIND_TRANSACTION(TID, option)
```

The option could be either 'CASCADE' (default), in which we perform a reverse operation to recover the state of records that are inserted/updated/deleted by the transaction with ID = TID and its dependent transactions recursively, or 'NON_CASCADE', in which if there do not exist dependent transactions, we perform a reverse operation to recover the state of the affected records.

## 3.3 Temporal Constraints

For **valid-time data model**: (1) The entity integrity is relaxed to the case that no intersection exists between valid time of any two records with the same primary key, e.g.,

```
PRIMARY KEY (ID, VT WITHOUT OVERLAPS)
```

(2) For reference integrity, there must exist one matching record in the parent table whose valid time contains the the valid time of the child record. For **transaction-time data model**: (1) constraints can only be added to the current records and follow the same logics in the conventional DBMSs. (2) users are not allowed to assign/change the value of transaction time and IDs, which can only be assigned/updated by the database system. (3) users are not allowed to change the historical records. For **bi-temporal data model**, the constrains are the combination of the valid-time data model and the transaction-time data model.

## 4. SYSTEM OVERVIEW

In this section, we outline the overall system architecture of TDSQL's temporal implementation. We also briefly present the functionality of its main components and will elaborate them in the next section. For ease of illustration, we use TDSQL and its temporal implementation interchangeably when the context is clear.

TDSQL supports temporal features mainly based on the extensions of three components shown in Figure 1, (1) parser, (2) query executor and (3) storage engine. Since the query executor relies on the storage engine, we introduce the extensions in the order of (1)(3)(2).

• **Parser.** We extend the parser to support the syntax of temporal queries, translate temporal queries into a simpler hybrid non-temporal and temporal queries, and output the translated syntax tree to the query optimizer. It has two main tasks. The first task is to perform the lexical and syntax analysis of the input temporal queries, which follow the SQL standard defined in SQL:2011. The other task is to translate temporal qualifiers, perform semantic check and output a syntax tree. In particular, valid-time involved operations are translated into equivalent non-temporal operations, as described in Section 3.2, while transaction-time involved operations remains unchanged in the syntax tree. Note transaction-time involved operations are implemented as a native support in TDSQL which will be discussed later. For illustration purposes, we give an example on the translation of a given temporal query below.

EXAMPLE 2. *Suppose R is an account balance relation. Consider the following temporal SQL statement that retrieves the account balance of player James on 2018-10-30, recorded in DBMS at 2018-10-11 00:00:00.*

```
SELECT ID, Player, Bal FROM R
WHERE Player = 'James'
  AND VT CONTAINS DATE '2018-10-30'
  FOR TT AS OF TIMESTAMP '2018-10-11 00:00:00'
```

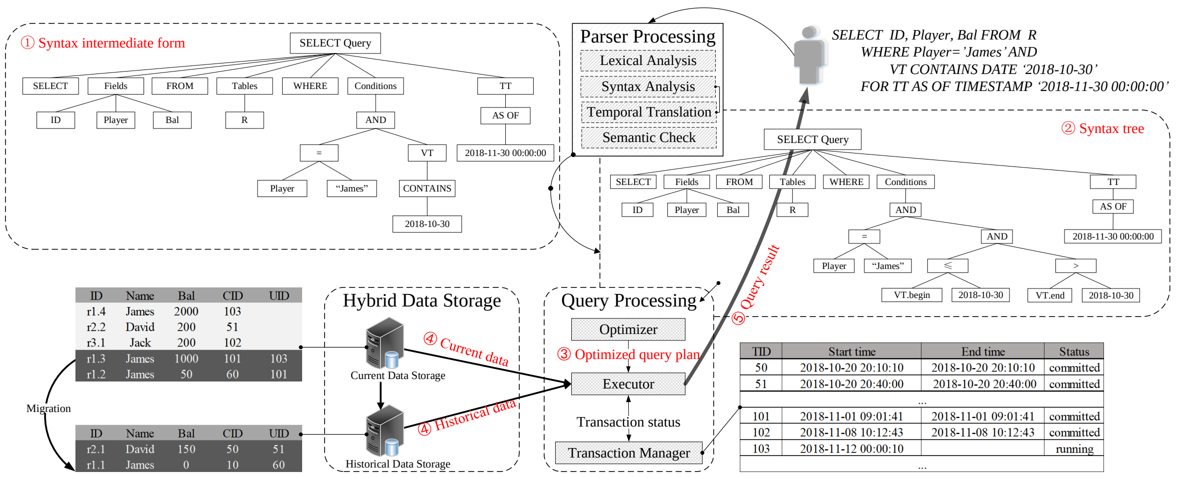
*The parser will translate the valid-time involved operations, which is underlined in the above statement, into equivalent non-temporal operations which is underlined in the following statement:*

```
SELECT ID, Player, Bal FROM R
WHERE Player = 'James'
  AND VT.st ≤ DATE '2018-10-30'
  AND VT.ed > DATE '2018-10-30'
  FOR TT AS OF TIMESTAMP '2018-10-11 00:00:00'
```

*For illustration purposes, we demonstrate the syntax tree and its intermediate form in Figure 1 for a given query.* □

• **Storage engine**. Like other temporal implementations in conventional DBMSs, TDSQL stores historical and current data separately in which we implicitly build a historical table to store the historical data for a transaction-time table, Nevertheless, it adopts a completely different way to maintain temporal data in terms of two key mechanisms, i.e., (1) when to transfer data from current tables to historical tables, and (2) how to organize the historical data considering that the historical data is ever-growing. We remain the latter to be explained in the next section.

Existing temporal implementations process transactional tasks on historical and current data in a synchronous mode.



*SELECT ID, Player, Bal FROM R*
*WHERE Player='James' AND*
*VT CONTAINS DATE '2018-10-30'*
*FOR TT AS OF TIMESTAMP '2018-11-30 00:00:00'*

| ID | Name | Bal | CID | UID |
|---|---|---|---|---|
| r1.4 | James | 2000 | 103 | |
| r2.2 | David | 200 | 51 | |
| r3.1 | Jack | 200 | 102 | |
| r1.3 | James | 1000 | 101 | 103 |
| r1.2 | James | 50 | 60 | 101 |

| ID | Name | Bal | CID | UID |
|---|---|---|---|---|
| r2.1 | David | 150 | 50 | 51 |
| r1.1 | James | 0 | 10 | 60 |

| TID | Start time | End time | Status |
|---|---|---|---|
| 50 | 2018-10-20 20:10:10 | 2018-10-20 20:10:10 | committed |
| 51 | 2018-10-20 20:40:00 | 2018-10-20 20:40:00 | committed |
| ... | | | |
| 101 | 2018-11-01 09:01:41 | 2018-11-01 09:01:41 | committed |
| 102 | 2018-11-08 10:12:43 | 2018-11-08 10:12:43 | committed |
| 103 | 2018-11-12 00:00:10 | | running |
| ... | | | |

**Figure 1: System Overview**

Any update/delete of a current record will produce one or multiple historical records which are then transferred from the current table to the historical table simultaneously. To do this, existing temporal implementations require to insert the new current record into the current table, and migrate the newly generated historical records from current table to the historical table in the same transaction.

While TDSQL processes transactional tasks on historical and current data in an asynchronous mode, we propose a hybrid data storage system encapsulated with a novel late data migration strategy, i.e., upon any update/delete of a current record will not cause an immediate data migration of the newly generated historical records. Instead, all newly generated historical records are migrated to the historical table when the database system starts to reclaim the storage occupied by records that are deleted or obsoleted. Storage reclaim in TDSQL is periodically invoked by the system in order to improve the efficiency of storage space and query performance. Similar operations can be found in other DBMSs, like PURGE in MySQL and VACUUM in PostgreSQL. Our late data migration strategy does not cause any omissions of historical records based on the facts that (1) an update/delete of a record in the conventional DBMSs will not physically remove it from its table, and (2) these deleted or obsoleted records are maintained in the rollback segment (similar to UNDO log in MySQL and Oracle) of the system. During the vacuum stage, we copy all newly generated historical records from the rollback segment and import them into the historical table in batch. As compared to the existing work, this late data migration brings two advantages. (1) Data migration in batch eliminates the access to historical table during the update/delete of current records, and hence reducing the transaction latency. (2) Conflicts in concurrent data access to historical tables are completely avoided, and hence result in a significant improvement of the transaction throughput for the whole system.

• **Query optimizer and executor**. Query optimizer takes a translated syntax tree as the input and generates the query execution plan for the executor. Except the transaction-time qualifiers, the translated syntax tree is optimized just like its non-temporal counterpart by the optimizer. We provide a native support for the execution of temporal queries with transaction-time qualifiers. Executor recognizes transaction-time qualifiers in the query execution plan, and invokes a native function call, including the tasks: (1) retrieving current and historical data of interest, and (2) integrating the query result and returning the result to the client.

Like other conventional DBMSs, the native TDSQL does not maintain the commit time of any transaction with each record. However, according to our new temporal model, each record $r$ needs to explicitly maintains the commit time of the transaction that creates/deletes $r$. To help set the commit time, we build an efficient transaction status manager. The manager maintains the status for the transactions, including commit time, in a transaction log. It helps efficiently retrieve the commit time for a given transaction ID. Our special design of the manager makes this retrieval at most one I/O cost. During the data migration, we update their transaction time for each of newly generated historical records based on its transaction ID. For the current records and historical records that have not yet been transferred to the historical table, we are still able to obtain the transaction time based on their transaction IDs via the manager.

Because we use current/historical tables to store current/historical records, respectively, the executor processes the query plan over the current table and historical table separately. To fetch the current records of interest, we simply execute the query plan over the current table. However, it is not trivial to retrieve the historical records of interest. Remind in our temporal data storage system, due to the late data migration, historical records maintained in the historical table is incomplete, and querying the historical table could return incomplete result. Consider that the remaining historical records are still in the current table, but are not visible to users. We propose a MVCC-based visibility check approach to retrieving historical records of interest. Details of the approach are elaborated in Section 5.2.

## 5. STORAGE AND QUERY PROCESSING

In this section, we elaborate two core techniques, temporal data storage and temporal query processing.

### 5.1 Temporal Data Storage

As discussed in the previous section, we store historical data and current data separately, and propose a late data migration strategy to transfer newly generated historical data in batch from the current table to the historical table.

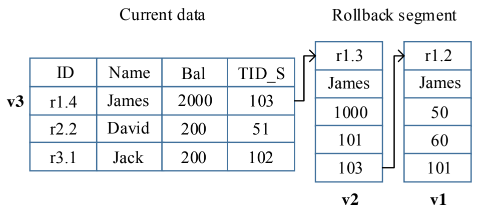


**Figure 2: A multi-versioned record in TDSQL**

As our data migration relies on the conventional storage engine, in this part, we first review how the storage system works in TDSQL, then present the historical data storage and its optimization, and finally discuss the implementation of the late data migration.

### 5.1.1 Current Data Storage

Like many other conventional DBMSs, including Oracle and MySQL, TDSQL is MVCC-based. That is, in TDSQL, when a record is updated, a new version of the record is created and inserted into the table, while the previous version is deleted and moved to the rollback segment. The newly generated record always maintain a link to its previous version (if any). According to the principle of MVCC, select/update/delete queries always locate the latest version of the record, and follow its link recursively to find a proper version based on the snapshot isolation mechanism [7, 26, 23]. Apparently, as compared to the temporal counterpart, in a non-temporal relation, the latest version of a record corresponds to the current record, and its previous versions correspond to historical records. A previous version will not be physically removed until the following conditions satisfy: (1) transactions that read/write this version are either committed or aborted; (2) a vacuum cleaner thread starts to garbage collect expired/aborted versions. For illustration purposes, Figure 2 shows an example of a record with three versions (labeled as $v1, v2, v3$) that are linked. Note its first version $v0$ has been transferred to the historical table.

### 5.1.2 Historical Data Storage

Historical records are inserted into the historical table in batch. As defined in SQL:2011, historical records are maintained in an append-only mode, i.e., any update/delete to the historical table is not allowed. For this reason, the size of historical data is always increasing, potentially causing a prohibitively expensive storage overhead. To address this issue, we employ the currently popular key value store instead of conventional relation storage model as the historical data storage. To do this, we can only maintain data changes for each base record, and hence reduce the size of storage space.

As defined in SQL:2011, an update of attribute value(s) of a record in a temporal relation could produce one or multiple historical records. Consider that these records are in fact the different versions of the same entity, and share the same values over the majority of attributes. We then focus on the storage space optimization by eliminating the maintenance for duplicated attribute values. Our main idea is to maintain the complete attribute values of the first version, and for any subsequent updates, only data changes are maintained so that redundant information storage are completely avoided. Take player James shown in Figure 2 for example. $r_{1.1}$ is the first version of James depicted in the balance table, and its attribute values are completely maintained in the historical data storage. $r_{1.2}$ is the second

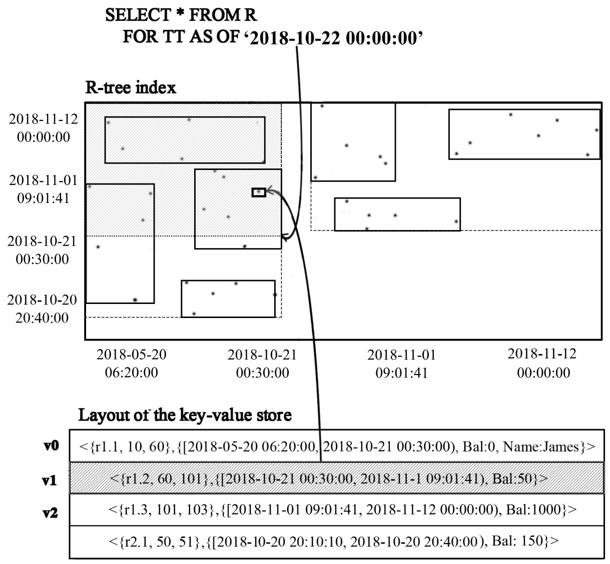

SELECT * FROM R
FOR TT AS OF '2018-10-22 00:00:00'
R-tree index
2018-11-12 00:00:00
2018-11-01 09:01:41
2018-10-21 00:30:00
2018-10-20 20:40:00
2018-05-20 06:20:00
2018-10-21 00:30:00
2018-11-01 09:01:41
2018-11-12 00:00:00
Layout of the key-value store
v0
<{r1.1, 10, 60},{[2018-05-20 06:20:00, 2018-10-21 00:30:00), Bal:0, Name:James}>
v1
<{r1.2, 60, 101},{[2018-10-21 00:30:00, 2018-11-1 09:01:41), Bal:50}>
v2
<{r1.3, 101, 103},{[2018-11-01 09:01:41, 2018-11-12 00:00:00), Bal:1000}>
<{r2.1, 50, 51},{[2018-10-20 20:10:10, 2018-10-20 20:40:00), Bal: 150}>

**Figure 3: Layout of the key value store with $R$-tree**

version by changing the balance to 50, and hence only attribute value 50 is stored. We store each historical record as a key value pair. The key consists of the value of the primary key and transaction IDs that a record associates. The value consists of the transaction time and either the attribute values for the first version, or the changed values with the changed attributes for any subsequent versions. For illustration purposes, in Figure 3, we show the layout of the key value store that maintains the historical data shown in Table 1. Under our design, an access to an entire historical record as of a specified point in time will result in an assembly from its first version to the specified point in time. Because in our key value store, keys are sorted and clustered indexed, retrieval and assembly a complete historical record as a specified point in time is quite efficient. Besides, to boost the transaction-time queries without specifying the primary key, we build a secondary R-tree index, which is widely supported in conventional DBMSs, on two-dimensional spaces, i.e., the start time and the end time of the transaction time. For illustration purposes, Figure 3 also shows an example of a secondary R-tree index.

### 5.1.3 Data Migration

As discussed in Section 5.1.1, newly generated historical records are maintained in the rollback segment, and have not yet transferred to the historical table. In TDSQL, a vacuum cleaner thread is periodically invoked to garbage collect expired/aborted versions and physically remove them from the rollback segment. To provide an automatic migration, we modify the logics of the vacuum cleaner thread by copying the historical records in the rollback segment to the historical table in batch. Our implementation is said to be lightweight and almost non-invasive in that we do not introduce new modules and simply copy the deleted/obsolete versions, which is supposed to be physically removed by the vacuum cleaner, to the historical table.

Algorithm 1 gives the details to migrate the historical data from the rollback segment to historical tables. We use variable *undos* to maintain the records to be migrated (line 1) and *kvs* to store historical records in *undos* into key-value format (line 2). Firstly, a coordinate thread is activated.

It scans data pages in rollback segment and collects all the records to be migrated, which are not locked by current running transactions (line 3–5). Subsequently, several worker threads will be invoked to transform historical records in *undos* into key/value pairs, which are then complemented with transaction commit time by searching it in the transaction manager (line 6–8). The key/value pairs *kvs* will be sent and stored in the key/value store in batch (line 9). Finally, the *truncate* function is invoked to clean up the rollback segment and exits (line 10).

**Algorithm 1:** migration(*rollback_segment*)

```
1  $undos \leftarrow \emptyset$; // unvacuumed undo records
2  $kvs \leftarrow \emptyset$; // key-value pairs for historical
     records
   // step#1:fetch rollback records
3  foreach $page \in rollback\_segment$ do
       // page.max_id:maximum transaction id
          operates this segment
       // S:the oldest snapshot that is still in
          use
4      if $page.max\_id < S.min$ then
5          $undos \leftarrow undos \cup getUndo(page)$;
   // step#2:transform
6  foreach $undo \in undos$ do
7      $kv = parse2KV(undo)$;
8      $kvs \leftarrow kvs \cup \{kv\}$;
   // step#3:call key-value pairs insert
      protocol
9  historical_storage::put($kvs$);
   // step#4:cleanup
10 truncate($rollback\_segment$);
```

## 5.2 Temporal Query Processing

In the query executor, as mentioned before, we rewrite the valid-time queries into conventional queries while remaining the transaction-time qualifiers unchanged in the query plan. Thus, in this section we mainly focus on the transaction-time query processing with the objective to efficiently retrieve current and historical data of interest, given that they are stored separately. Unless otherwise specified, a relation/table mentioned in this section is referred to as a transaction-time relation/table.

*Processing transaction-time queries is not trivial because the transaction time in the data model is difficult to assign and update.* Recall that each record $r$ associates a transaction time $TT$. Theoretically, when $r$ is created by a transaction $\mathcal{T}_s$, its $TT.st$ should be assigned to the commit time of $\mathcal{T}_s$. Similarly, when $r$ is deleted/updated by another transaction $\mathcal{T}_e$, its $TT.ed$ should be updated to the commit time of $\mathcal{T}_e$. The reason is that based on the snapshot isolation theorem [7, 26], given a record $r$, and a transaction $\mathcal{T}$, whether $r$ is visible to $\mathcal{T}$, i.e., whether $\mathcal{T}$ is able to read $r$, only if the start time of $\mathcal{T}$ is after the commit time of the transaction that creates $r$, and before the commit time of the transaction that updates/deletes $r$. To guarantee satisfactory performance of the whole system, conventional DBMSs, including TDSQL, do not maintain the commit time of the transaction that creates/updates/deletes with each record. In practice, it is also inefficient to set and update the transaction time for all inserted/deleted/updated records upon a transaction is committed. Interestingly, SQL:2011 leaves the transaction time up to SQL-implementations to pick an appropriate value for the transaction timestamp of a transaction. While many temporal implementations in conventional DBMSs, either pick up the start time of the transaction, or the time of the operation that inserts/updates/deletes the record, we argue that this could potentially cause an incorrect result based on the snapshot isolation theorem. For this reason, we first propose a transaction status manager based on which transaction times for records can correctly be obtained. We then develop an MVCC-based visibility check approaches based on which the transaction-time queries can be correctly answered.

### 5.2.1 Transaction Time Maintenance

We propose a transaction status manager that maintains the status for transactions, including commit time, a running/committed/aborted state in a transaction log. Given a transaction $\mathcal{T}$, we insert the status of $\mathcal{T}$ into the manager when $\mathcal{T}$ starts to execute, and update the commit time, state of $\mathcal{T}$ upon $\mathcal{T}$ is committed/aborted. Note that we do not necessarily maintain the transaction IDs. This is because that the length of each status data structure is fixed, and we store the status of each transaction in the offset of the transaction log by multiplying the transaction ID and the status length. Hence, retrieving the status of each transaction by a given transaction ID requires at most one I/O cost.

Based on the proposed transaction manager, records are able to associate the transaction time implicitly or explicitly. On one hand, during the data migration, for each newly generated historical record $r$ in the rollback segment, via the transaction ID associated with $r$, we replace its transaction time with the commit time that is maintained in the transaction status manager. Thus, we can explicitly maintain the transaction time with records in the historical table. On the other hand, for the records in the current table or in the rollback segment, although they do not associate the transaction time, it is able to obtain the transaction time based on the transaction status management by providing their transaction IDs.

### 5.2.2 Record Visibility Check

A non-temporal query in either TDSQL or other conventional DBMSs are almost snapshot based. A snapshot is created when a transaction $\mathcal{T}$ starts to execute, and in essence, it logically stores all latest versions that are inserted/updated/deleted by the transactions that have been committed before $\mathcal{T}$ starts. Thus, a non-temporal query is to retrieve records of interest that are logically stored in the snapshot. In other words, given a snapshot $\mathcal{S}$, records that are readable in $\mathcal{S}$ are returned as the query result if they satisfy the requirements specified in the query conditions.

As presented in Section 3.2, the semantics of a transaction-time query is to retrieve records that are current either as of a specified point $s$ in time (namely time-travel query) or between any two points $s, t$ in time (namely time-slice query). Correctly answering time-travel queries requires to reconstruct the snapshot at time $s$, and retrieve all records that are readable in this snapshot and satisfy the requirements specified in the query conditions. Correctly answering time-

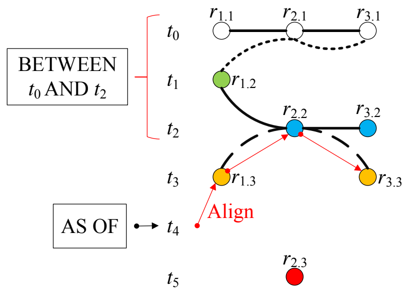


**Figure 4: Transaction-time query processing**

slice query requires to reconstruct all the snapshots during the period $[s, t]$, and retrieve all records that are readable in these snapshots and satisfy the requirements specified in the query conditions. For illustration purposes, we give an example to show how a transaction-time query works in Example 3.

Example 3. *Suppose there exist three records,* $r_{1.1}$, $r_{2.1}$ *and* $r_{3.1}$ *in a transaction-time table, shown in Figure 4. First, a transaction* $\mathcal{T}_1$ *committed at time* $t_1$ *updates* $r_{1.1}$ *to* $r_{1.2}$*. Subsequently, another transaction* $\mathcal{T}_2$ *committed at time* $t_2$ *updates* $r_{2.1}$ *to* $r_{2.2}$*, and* $r_{3.1}$ *to* $r_{3.2}$*. Next, transaction* $\mathcal{T}_3$ *committed at time* $t_3$ *updates* $r_{1.2}$ *to* $r_{1.3}$*, and* $r_{3.2}$ *to* $r_{3.3}$*; transaction* $\mathcal{T}_4$ *committed at* $t_5$ *updates* $r_{2.2}$ *to* $r_{2.3}$*. Suppose we issue a time-travel query at time* $t_4$*. As discussed above, we need to reconstruct the snapshot at time* $t_4$*, and all records that are readable in this snapshot are returned, i.e.,* $r_{1.3}, r_{2.2}, r_{3.3}$*. If we issue a time-slice query between* $t_0$ *and* $t_2$*, then* $r_{1.1}, r_{2.1}, r_{3.1}, r_{1.2}, r_{2.2}, r_{3.2}$ *are returned.* □

Retrieval of records with multiple versions from historical data storage is straightforward due to the explicitly maintained transaction commit time. We then focus on the current data storage which include current table and rollback segment, as discussed in Section 5.1.1. Thus, our objective is to fetch versions of interest from both parts. We propose a snapshot based approach to check whether a version can be visible to a given temporal query. First, each transaction will be assigned with an unique snapshot when the transaction starts. The snapshot $S$ includes the following four variables: (1) *S.tids* includes all active transactions' IDs when a snapshot is generated; (2) *S.min* represents the minimum transaction ID in *S.tids*; (3) *S.max* is the first unassigned transaction ID; (4) *S.creator* is the snapshot owner's transaction ID.

Algorithm 2 shows how to fetch versions of interest in the current data storage. The pseudo-code from line 2 to line 12 is to answer as_of queries while line 13–21 is to answer time-period queries. Due to the space limitations, we merely elaborate the as_of query processing as time-period query processing follows a similar way. We sequentially scan items in *Candidates* that maintains records qualifying the non-temporal conditions (line 3). For each record, we will first check whether the latest version is visible in this transaction's $S$ (line 5–7) and the temporal query constraints (8–12). If either of them is violated, the previous version will be taken out (line 6, 12) and continues to check on it.

Algorithm 3 shows the temporal constraints for queries on current storage. Since data in current storage maintains the transaction time in transaction log, we should get the commit time of *v.cid* and *v.uid* from transaction log (line 2–3).

**Algorithm 2:** currentStorageRead($S$,$\mu$))

```
input  : snapshot $S$, query type $\mu$
output: versions $\varphi$ meet the temporal condition
1  $\varphi \leftarrow \emptyset$;
2  if $\mu$=as_of then
       // Candidates:records should be check
3      foreach $rec \in Candidates$ do
4          while $rec <> null$ do
5              if $rec.cid >= S.max \vee (rec.cid \in S.tids \wedge rec.cid <> S.creator)$ then
                   // fetch the previous version
6                  $rec \leftarrow prevVer(rec)$;
7                  continue;
8              if $temporalCheck(rec)$ then
9                  $\varphi \leftarrow \varphi \cup \{rec\}$;
10                 break;
11             else
12                 $rec \leftarrow prevVer(rec)$;
13 if $\mu$=from_to $\vee$ $\mu$=between_and then
14     foreach $rec \in Candidate$ do
15         while $rec <> null$ do
16             if $rec.cid >= S.max \vee (rec.cid \in S.tids \wedge rec.cid <> S.creator)$ then
                   // fetch the previous version
17                 $rec \leftarrow prevVer(rec)$;
18                 continue;
19             if $temporalCheck(rec)$ then
20                 $\varphi \leftarrow \varphi \cup \{rec\}$;
21             $rec \leftarrow prevVer(rec)$;
```

Then we follow the temporal semantics defined in SQL:2011 to examine whether version $v$ satisfies the temporal constraints (line 4–12).

**Algorithm 3:** *temporalCheck*($v$)

```
1  switch $\mu$ do
       // get commit time from trsancation log
2      $\gamma_{st} \leftarrow getCommitTime(v.cid)$;
3      $\gamma_{ed} \leftarrow getCommitTime(v.uid)$;
4      case as_of do
5          if $\gamma_{st} \leq ts \wedge \gamma_{ed} > ts$ then
6              return true;
7      case from_to do
8          if $\gamma_{st} < ts_2 \wedge \gamma_{ed} > ts_1$ then
9              return true;
10     case between_and do
11         if $\gamma_{st} \leq ts_2 \wedge \gamma_{ed} > ts_1$ then
12             return true;
13 return false;
```

## 6. IMPLEMENTATION

Although we have presented a built-in temporal support on TDSQL, our proposed techniques are also applicable to other MVCC-based DBMSs, like Oracle, MySQL, PostgreSQL.

In this section, we shall briefly discuss the temporal implementation of our key techniques on MySQL.

**System architecture.** MySQL's system architecture consists of two layers, server layer and storage engine layer. The former mainly takes charge of parsing, query processing and optimization. The latter is mainly responsible for transaction processing, data storage and retrieval. Consider InnoDB is the currently most popular storage engine for MySQL, all our discussion is based on MySQL/InnoDB.

**Table 2: Our implementation and major APIs**

| Return Type | Function | Description |
|---|---|---|
| void | Sql_cmd_dml::translate | Syntax translation |
| dberr_t | row_search_history | Retrieval function for historical data |
| dberr_t | row_search_current | Retrieval function for current data |
| ulint_t | row_purge_migrate | Data migration |

As discussed before, extensions of temporal query processing in TDSQL require to a modification of parser, query executor, storage engine, and transaction processing system. For reference, we list the main APIs that have been either modified or newly introduced in Table 2 .

• **Parser.** We extend the parser API to support the recognition of temporal qualifiers to ensure the correctness of syntax check. We also add a translator API, *Sql_cmd_dml::translate()*, to transform the valid-time involved operations to equivalent non-temporal operations in the parse tree.

• **Query executor.** We extend the executor API to recognize the transaction-time qualifiers. If there are no transaction-time qualifiers, then logics of the executor remain unchanged; otherwise, the query executor runs the query execution plan over the current table and the historical table separately. To retrieve a complete set of historical records of interest, we mainly introduce two new APIs: (1) *row_search_history()* is to extract records of interest from historical table, (2) *row_search_current()* is to extract historical records from current table according to the given temporal condition using the MVCC-based visibility check approach, depicted in Algorithm 2.

• **Storage engine.** We extend the storage engine so that it can manage both current data and historical data automatically. Our extension is based on the MySQL purge feature. An update/delete of a record in MySQL will not cause a physical remove. In contrast, all physical removes of deleted records are conducted in batch by the purge operation, which is periodically invoked by the system. To implement our late data migration, we modify the purge by adding an API *row_purge_migrate()* to transfer all newly generated historical data from current table to historical table in batch.

• **Transaction processing system.** MySQL/InnoDB does not maintain any transaction status of each record when a transaction is committed for the purposes of performance-critical aspect. While in our temporal data model, each record associates transaction time and IDs, it is necessary to extend the transaction processing system of InnoDB by assigning/updating each record with transaction time when a transaction is committed. We add a transaction manager that maintains and searches the status of all transactions. The manager consists of two modules: (1) a transaction log that is particularly designed to maintain the status, including commit time, running/committed/aborted state, for all transactions; (2) efficient data structures to query the status of a given transaction ID. Each record is assigned/updated with transaction IDs when it is generated/updated/deleted. Its transaction time is either retrieved or updated with a transaction ID via the manager if necessary. Hence, our temporal implementation is applicable to MySQL/InnoDB.

## 7. APPLICATIONS

In this section, we present three real temporal applications that have already adopted TDSQL.

• **Account reconciliation.** Account reconciliation is a core business in Tencent for payment and gaming services. It periodically examines the account balance with the expense statement for every customer/player. Once an account imbalance occurs, it is necessary to efficiently trace the transactions that possibly destroy the account balances. For illustration purposes, we show how to do account reconciliation in TDSQL using a single SQL statement below, in which *account* and *water* are two tables that maintain account balances and expense statements, respectively.

```
SELECT * FROM (
  account FOR TT FROM ts1 TO ts2 as A
  FULL OUTER JOIN
  account FOR TT FROM ts1 TO ts2 as B
  ON A.UID = B.CID
)
FULL OUTER JOIN
water FOR TT FROM ts1 TO ts2 as C
ON B.CID = C.UID
```

• **Logical data corruption recovery.** Logical data corruption could occasionally occur in some critical services, e.g., accidental deletion of one or multiple friends in WeChat or games, erroneous transfer by typing an incorrect bank account in the payment. Using the transaction ID query proposed in TDSQL, it is able to do fast and robust logical data corruption recovery. For example, an erroneous transaction with ID 123456 was made and successfully submitted by some operator. In TDSQL, it is convenient for users to perform reverse operations so that all affected account balances can be rewound using the following SQL statement.

```
REWIND_TRANSACTION(123456, 'CASACDE')
```

• **Cloud resource management.** Tencent cloud has hundreds thousands of storage and computing nodes, serving billions of users and millions of enterprises. Typical queries include how many nodes were/are in use given a point in time, given two points in time, details that a machine is actually used in this duration, the history of resource use of a given user. For example, the following SQL statement retrospects the usage of the server with ID 123456 during a historical period which is from 2018-01-01 00:00:00 to 2019-01-01 00:00:00.

```
SELECT ID, type, status, department
FROM servers
```

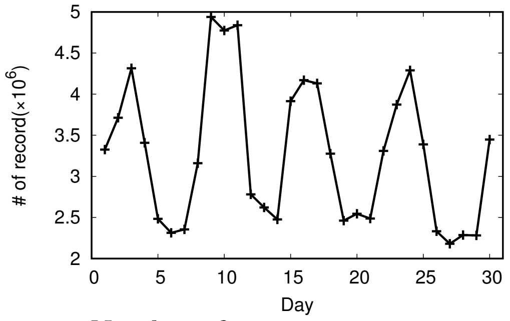


**Figure 5: Number of expense statements per day**

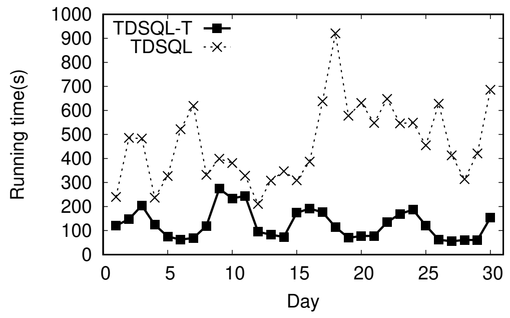


**Figure 6: Evaluation on Tencent-RB**

```
FOR TT FROM TIMESTAMP '2018-01-01 00:00:00'
TO TIMESTAMP '2019-01-01 00:00:00'
WHERE ID=123456
```

As discussed before, TDSQL stores current/historical data separately, and data migration from the current table to the historical table is periodically executed. One benefit, not mentioned before and brought by the late data migration, is that during the data migration we can transfer the historical data to another OLAP system. In this way, OLAP system is responsible for processing historical queries, and conventional TDSQL is still used to process transactional queries, thus avoiding the performance degradation of TDSQL by introducing the temporal support. From this perspective, our design makes the system become a hybrid transaction and analysis processing cluster (a.b.a HTAC) for current data and historical data managment.

## 8. EVALUATION

Our experiments are carried out on a server with an Intel 24-core Xeon 2.4GHz CPU, 128GB Memory, and a 12-disk RAID 0 Enterprise Performance 10K HDD drive, running a CentOS 6.2 operation system with kernel version 3.10.106. We compare TDSQL with Oracle 11g, SQL Server 2017, and MariaDB 10.3.11 in the experiments as these RDBMs provide temporal support. We use the default configurations for the systems that participate in the comparison, except that the buffer pool and the thread pool are set to 8GB and 48, respectively if the systems provide these two parameters.

We conduct the experiments using the following one real and two synthetic benchmarks.

• **Tencent-RB** is a real benchmark abstracted from the Tencent billing service platform. It has two relations. One relation $R$ depicts user account balances which are fairly stable with nearly 500 million of records. The other relation $W$ depicts the expense statements. We collect the recent one month expense statements and store them in $W$. For illustration purposes, we show the number of every single consumption per day of the month in Figure 5.

• **TPC-C** is a popular OLTP benchmark with a mixture of read-only and update intensive transactions that simulate the activities in order-entry and delivery, payment process, and stock monitoring. The major metric of TPC-C is tpmC, which is measured in the number of new-order transactions per minute. To support temporal data management, we add transaction time as an additional attribute to each of the nine relations in the benchmark. When a record is created/updated/deleted, the start/end time of its transaction time is set to the time when the transaction is committed.

• **TPC-BiH** [20] is a recently proposed benchmark, which is particularly designed for the evaluation over temporal databases. It builds on the solid foundations of TPC-H benchmark but extends it with a rich set of temporal queries and update scenarios that simulate real-life temporal applications from SAP's customer applications.

### 8.1 Evaluation on Tencent-RB

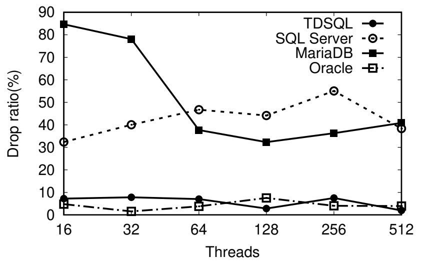
For ease of illustration, we use **TDSQL** and **TDSQL-T** to denote TDSQL without and with temporal support, respectively. We first run regular queries to do daily account reconciliation on TDSQL as usual and collect the query performance. As a comparison, we transform the regular query into equivalent temporal query, run it on TDSQL-T, and collect the query performance as well.

Figure 6 shows the result by comparing the performance of TDSQL and TDSQL-T. It can be observed that TDSQL-T is in general significantly faster than TDSQL, ranging from 1.32× to 9.12×, depending on the data size. In this one month duration, the performance of TDSQL-T is less sensitive when the data size varies, and is always better than TDSQL. The reason of this advantage in essence is that TDSQL-T deals with less data. TDSQL executes account reconciliation in two steps: (1) compute account changes by scanning the whole relation $R$, in which the number of records is about 500 million; (2) join all account changes with the expense statements in $W$, in which the number of records is shown in Figure 5. As TDSQL-T has already maintained the changed accounts automatically, of which the number varies from 2 million to 5 million, TDSQL-T only needs to join the changed accounts with $W$, which results in much less join computation than that of TDSQL.

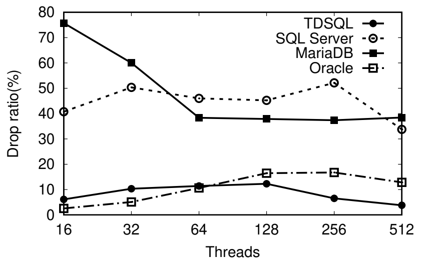
### 8.2 Evaluation on TPC-C

We study the effect by introducing the temporal support on TPC-C benchmark for the conventional DBMSs. Let $tpmC_{temporal}$ and $tpmC_{non-temporal}$ be the number of new-order transactions per minute by running TPC-C workload in conventional DBMSs with and without temporal support. Note that all regular queries in TPC-C retrieve current data, and we do not rewrite them to temporal DML queries. We introduce a new metric, namely performance drop ratio, which is defined as $1 - \frac{tpmC_{temporal}}{tpmC_{non-temporal}}$, to do the performance study.

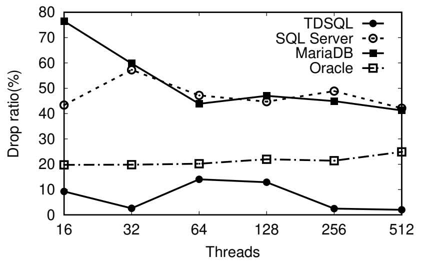
Figure 7 shows the effect on the drop ratios by varying the number of data warehouses. From the figures, we make



(a) 256 warehouses



(b) 512 warehouses



(c) 1024 warehouses

**Figure 7: Effect on the drop ratio by introducing temporal support**

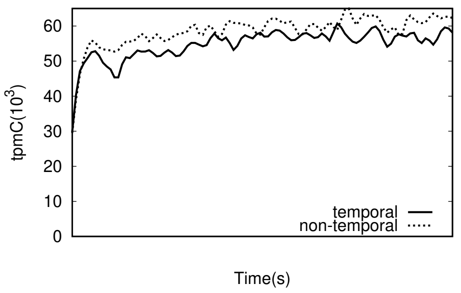


**Figure 8: Drop ratio & time**

two observations. First, the drop ratio of TDSQL is 7% on average, and varies from 2% to 14%, showing that TDSQL's temporal implementation is lightweight. Second, the drop ratio of TDSQL is comparable to that of Oracle when the number of data warehouses varies from 256 to 512. While the number of data warehouses reaches 1024, TDSQL shows its superiority, achieving 8% to 21% smaller drop ratio than Oracle. In all cases, the drop ratio of TDSQL is significantly smaller than that of MariaDB and SQL Server, further showing its lightweight feature. As repeatedly discussed before, TDSQL transfers the historical data in batch only during the garbage collection, which is periodically invoked by the system, while the other systems manipulate historical records and current records synchronously, and hence results in smaller drop ratios. To make a further performance study of TDSQL, we run TDSQL with and without temporal support in continuous 600 seconds, separately, and collect $tpmC_{temporal}$ and $tpmC_{non-temporal}$ every 10 seconds. We plot the result in Figure 8. Again, we can see that the drop ratio is stable between 2% to 9%.

## 8.3 Evaluation on TPC-BiH

We study the performance of various temporal implementations on TPC-BiH benchmark. The benchmark contains four categories of queries, namely time-travel query, pure-key query, bi-temporal query, and range-timeslice query. Each category of queries contains 5 to 10 queries. We set the scale factor to 1 and make 100,000 updates in TPC-BiH.

Figures 9(a)(b)(c)(d) illustrate the execution time of running TPC-BiH on the comparative systems. For answering time-travel queries that retrieve records in a given point in time, it can be observed that on average TDSQL runs 1×, 4×, 29× faster than Oracle, SQL Server, MariaDB, respectively. In particular, TDSQL runs constantly faster than others in all queries except Q1, Q2, and Q4, in which Oracle performs the best, followed by TDSQL, SQL Server, and MariaDB. Interestingly, for answering pure-key queries that retrieve all records with the same keys, i.e, all current and historical versions of the same entity, SQL Server performs the best, followed by TDSQL, Oracle, and MariaDB. For answering bi-temporal queries with both valid time and transaction time, and answering range-timeslice queries that retrieve records with transaction time locating between two points in time, as we can see, the performance follows similar trends of that in answering time-travel queries. The reason, as discussed in Section 5, is that various optimizations are applied to the temporal query processing including an efficient key value store, with a clustered index and a secondary $R$-tree index built on the transaction time.

In summary, besides an enriched expressiveness, TDSQL can achieve better query performance in many applications, like account reconciliation. Compared with other temporal database systems, TDSQL almost has the minimal performance loss by introducing the temporal features, and performs the best for most of the temporal queries.

## 9. CONCLUSION

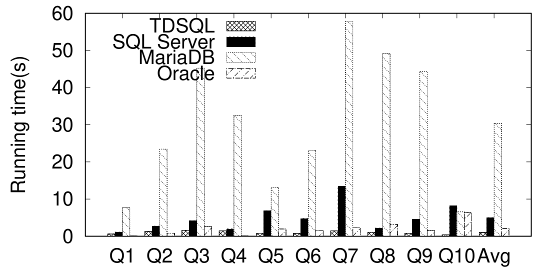
In this paper, we present a lightweight yet efficient built-in temporal implementation in TDSQL. Our implementation not only supports the temporal features defined in SQL:2011, but also makes an extension of the temporal model. We propose a novel late data migration strategy to manage current data and historical data in a very lightweight way. We also develop a native operator to support transaction-time queries with various optimizations. Extensive experiments are conducted on both real and synthetic TPC benchmarks, and the results show TDSQL almost has the minimal performance loss by introducing the temporal features, and performs the best for most of the temporal queries.

## 10. ACKNOWLEDGMENTS

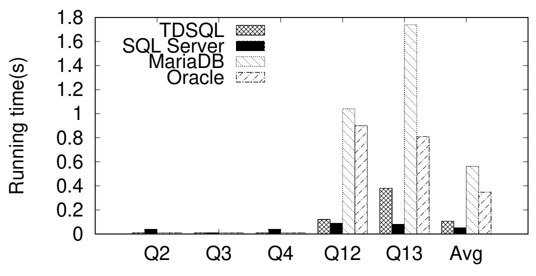
This work was partially supported by National Key Research and Development Program of China under Grant 2018YFB10044401, National Natural Science Foundation of China under Grant 61732014, Beijing Municipal Science and Technology Project under Grant Z171100005117002, Tencent Rhino-Bird Joint Research Program.

## 11. REFERENCES

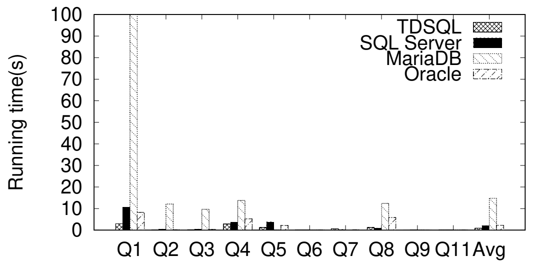
[1] DB2. `https://www.ibm.com/analytics/us/en/db2`.
[2] Mariadb. `https://mariadb.com/kb/en/library/system-versioned-tables/`.
[3] Oracle. `https://www.oracle.com`.
[4] PostgreSQL. `https://www.postgresql.org`.
[5] SQL Server. `https://docs.microsoft.com/en-us/sql/relational-databases/tables/temporal-tables`.



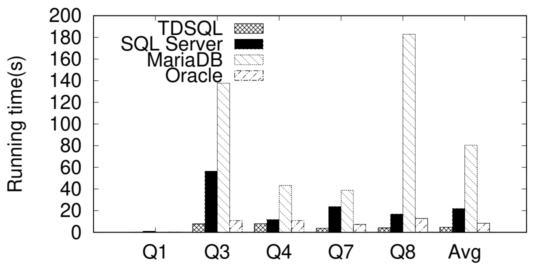
(a) Time-travel Query



(b) Pure-key Queries



(c) Bi-temporal Query



(d) Range-timeslice Query

**Figure 9: Performance study on TPC-BiH**

[6] M. Al-Kateb, A. Ghazal, A. Crolotte, R. Bhashyam, J. Chimanchode, and S. P. Pakala. Temporal query processing in teradata. In *Joint 2013 EDBT/ICDT Conferences, EDBT '13 Proceedings, Genoa, Italy, March 18-22, 2013*, pages 573–578, 2013.

[7] H. Berenson, P. A. Bernstein, J. Gray, J. Melton, E. J. O'Neil, and P. E. O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD Conference*, pages 1–10. ACM Press, 1995.

[8] M. Böhlen and C. Jensen. *Temporal Data Model and Query Language Concepts*, pages 437–453. 12 2003.

[9] C. X. Chen and C. Zaniolo. Sql$^{st}$: A spatio-temporal data model and query language. In *ER*, volume 1920 of *Lecture Notes in Computer Science*, pages 96–111. Springer, 2000.

[10] J. Chomicki. Efficient checking of temporal integrity constraints using bounded history encoding. *ACM Trans. Database Syst.*, 20(2):149–186, 1995.

[11] J. Clifford, C. E. Dyreson, R. T. Snodgrass, T. Isakowitz, and C. S. Jensen. "now". In *The TSQL2 Temporal Query Language*, pages 383–392. 1995.

[12] R. Elmasri and G. T. J. Wuu. A temporal model and query language for ER databases. In *ICDE*, pages 76–83. IEEE Computer Society, 1990.

[13] S. K. Gadia and C. Yeung. A generalized model for a relational temporal database. In *SIGMOD Conference*, pages 251–259. ACM Press, 1988.

[14] C. S. Jensen, J. Clifford, R. Elmasri, S. K. Gadia, P. J. Hayes, and S. Jajodia. A consensus glossary of temporal database concepts. *SIGMOD Record*, 23(1):52–64, 1994.

[15] C. S. Jensen, J. Clifford, S. K. Gadia, A. Segev, and R. T. Snodgrass. A glossary of temporal database concepts. *SIGMOD Record*, 21(3):35–43, 1992.

[16] C. S. Jensen, C. E. Dyreson, M. H. Böhlen, J. Clifford, R. Elmasri, S. K. Gadia, F. Grandi, P. J. Hayes, S. Jajodia, W. Käfer, N. Kline, N. A. Lorentzos, Y. G. Mitsopoulos, A. Montanari, D. A. Nonen, E. Peressi, B. Pernici, J. F. Roddick, N. L. Sarda, M. R. Scalas, A. Segev, R. T. Snodgrass, M. D. Soo, A. U. Tansel, P. Tiberio, and G. Wiederhold. The consensus glossary of temporal database concepts - february 1998 version. In *Temporal Databases, Dagstuhl*, pages 367–405, 1997.

[17] C. S. Jensen and R. T. Snodgrass. Temporal data management. *IEEE Trans. Knowl. Data Eng.*, 11(1):36–44, 1999.

[18] L. Jiang, B. Salzberg, D. B. Lomet, and M. B. García. The bt-tree: A branched and temporal access method. In *VLDB*, pages 451–460. Morgan Kaufmann, 2000.

[19] M. Kaufmann, P. M. Fischer, N. May, C. Ge, A. K. Goel, and D. Kossmann. Bi-temporal timeline index: A data structure for processing queries on bi-temporal data. In *ICDE*, pages 471–482. IEEE Computer Society, 2015.

[20] M. Kaufmann, P. M. Fischer, N. May, A. Tonder, and D. Kossmann. Tpc-bih: A benchmark for bitemporal databases. In *TPCTC*, volume 8391 of *Lecture Notes in Computer Science*, pages 16–31. Springer, 2013.

[21] K. G. Kulkarni and J. Michels. Temporal features in SQL: 2011. *SIGMOD Record*, 41(3):34–43, 2012.

[22] A. Kumar, V. J. Tsotras, and C. Faloutsos. Access methods for bi-temporal databases. In *Temporal Databases*, Workshops in Computing, pages 235–254. Springer, 1995.

[23] H. Li, Z. Zhao, Y. Cheng, W. Lu, X. Du, and A. Pan. Efficient time-interval data extraction in mvcc-based rdbms. *World Wide Web*, 2018.

[24] D. B. Lomet, R. S. Barga, M. F. Mokbel, G. Shegalov, R. Wang, and Y. Zhu. Transaction time support inside a database engine. In *ICDE*, page 35. IEEE Computer Society, 2006.

[25] N. A. Lorentzos and Y. G. Mitsopoulos. SQL extension for interval data. *IEEE Trans. Knowl. Data Eng.*, 9(3):480–499, 1997.

[26] D. R. K. Ports and K. Grittner. Serializable snapshot isolation in postgresql. *PVLDB*, 5(12):1850–1861, 2012.

[27] A. P. Sistla and O. Wolfson. Temporal conditions and integrity constraints in active database systems. In *SIGMOD Conference*, pages 269–280. ACM Press, 1995.

[28] M. Stonebraker, E. Wong, P. Kreps, and G. Held. The design and implementation of INGRES. *ACM Trans. Database Syst.*, 1(3):189–222, 1976.

[29] Y. Tao and D. Papadias. Mv3r-tree: A spatio-temporal access method for timestamp and interval queries. In *PVLDB*, pages 431–440. Morgan Kaufmann, 2001.

[30] Y. Tao, D. Papadias, and J. Sun. The tpr*-tree: An optimized spatio-temporal access method for predictive queries. In *VLDB*, pages 790–801. Morgan Kaufmann, 2003.

[31] J. R. R. Viqueira and N. A. Lorentzos. SQL extension for spatio-temporal data. *VLDB J.*, 16(2):179–200, 2007.