T-SQL: A Lightweight Implementation to Enable Built-in Temporal Support in MVCC-based RDBMSs

Zhanhao Zhao, Wei Lu, Hongyao Zhao, Zongyan He, Haixiang Li, Anqun Pan, Xiaoyong Du

Abstract—The adoption of temporal expressions into SQL:2011 has continuously driven the extensions of temporal support in relational database systems (a.b.a. RDBMSs). In this paper, we present *T-SQL*, a lightweight yet efficient built-in temporal implementation in RDBMSs. T-SQL entirely relies on multi-version concurrency control (MVCC), widely adopted in RDMBSs, to manage *temporal data*. For temporal data, *current records* are maintained in legacy databases, and *historical records*, i.e., previous versions of current records (if any), which used to be periodically reclaimed are separately maintained in KV stores. To enable temporal query processing under SQL:2011, we extend the query engine in legacy RDBMSs to support query processing over either current records or historical records or both. Further, regarding temporal data are ever-increasing, we propose various optimizations to reduce the storage overhead of KV stores while keeping efficient query performance. We elaborate a publicly available implementation on how to integrate T-SQL into both centralized and distributed RDBMSs. We conduct extensive experiments on both YCSB and TPC-series benchmarks by comparing T-SQL with other temporal database systems. The results show that T-SQL is both lightweight and efficient.

Index Terms—RDBMS, Temporal Database, MVCC, SQL:2011, KV Store

1 INTRODUCTION

It is of great importance to maintain not only currently valid data, but also the history of all data changes. Typical examples include forensic analysis and legal requirements to store data for a certain number of years, retrospective and trend data analysis to offer the asset use details as of two years ago, and logical corruption recovery to rewind tuples/relations/databases as of a particular point in time [32].

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 RDBMSs 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 start to provide built-in temporal support. The temporal extensions typically include: (1) extension of non-temporal relations to temporal relations, (2) 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 relation definitions, temporal queries, and others (e.g., temporal constraints). As compared to the non-temporal counterpart, a temporal relation 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.

TABLE 1
Gaming player account balance

ID	Player	Bal	Valid Time	Transaction Time
$r_{1.1}$	James	0	[2018-05-20,2018-10-21)	[2018-05-20,2018-10-21)
$r_{1.2}$	James	50	[2018-10-21,2018-11-01)	[2018-10-21,2018-11-01)
$r_{1.3}$	James	1000	[2018-11-01,2018-11-12)	[2018-11-01,2018-11-12)
$r_{1.4}$	James	2000	[2018-11-12,2019-05-11)	[2018-11-12,∞)
$r_{2.1}$	David	150	[2018-10-20,2018-10-20)	[2018-10-20,2018-10-20)
$r_{2.2}$	David	200	[2018-10-20,2019-04-19)	[2018-10-20,∞)
$r_{3.1}$	Jack	200	[2018-11-08,2019-05-07)	[2018-11-08,∞)

Example 1. *Table 1 shows an example of the account balance for gaming players. Each player has one current record and perhaps one or several historical records. For example, $r_{1.4}, r_{2.2}$, $r_{3.1}$ are current records of players James, David, and Jack, respectively, while the remaining are the historical records of either James or David. Assume each recharge increases the validity of account balance by 6 months, e.g., the valid time of current record $r_{1.4}$ is [2018-11-12,2019-05-11). Besides, the transaction time of each record r indicates the timestamps on which r is created and deleted/updated if any (otherwise the timestamp is set to ∞). For instance, it can be observed that $r_{1.4}$ is created on '2018-11-12' and has not yet been deleted/updated.* □

Thus far, we have witnessed a big burst of temporal support

• *Zhanhao Zhao, Wei Lu, Hongyao Zhao, Zongyan He, and Xiaoyong Du are with the Key Laboratory of Data Engineering and Knowledge Engineering, Ministry of Education, China, and School of Information, Renmin University of China, China.*
E-mail: {zhanhaozhao,lu-wei,hongyaozhao,zongyanhe,duyong}@ruc.edu.cn
• *Haixiang Li and Anqun Pan are with Tencent Inc., China.*
E-mail: {blueseali,aaronpan}@tencent.com

in legacy database systems, including Oracle [4], Teradata [8], MariaDB [3], SQL Server [6]. However, they suffer from either limited expressiveness or poor performance, or both. *First, the temporal data model is inadequate.* It is often reported that DBAs are fired or forced to run away due to an accidental deletion of valuable business data. This kind of unexpectedly committed transactions, namely logical data corruption, is difficult to avoid in real applications. Yet, the temporal data model defined in SQL:2011 does not explicitly support logical data corruption recovery. Technically, to address this issue, it is necessary to extend the temporal data model, based on which we first figure out the erroneous transaction, and then apply a sequence of reverse operations over the records involved in the erroneous transaction. *Second, the performance of legacy databases suffers from the synchronization overhead by introducing the built-in temporal support*. Existing temporal implementations store history and current records separately so as to skip over historical records for current data query processing, which is deemed to the dominant temporal queries. However, this separation degrades the throughput of conventional transactional workloads because any update/delete of a record r will cause a synchronization that transfers r from the current relation to the historical relation. In this way, any update/delete of a record in the current relation will require exclusive access to the historical relation, leading to a significant drop in the whole system's throughput. *Third, temporal data are maintained in an append-only mode, causing an ever-increasing size.* The overhead of maintaining a large volume of temporal data degrades the query performance, and hence it is of great necessity to develop a scalable storage engine to enable efficient query processing.

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

- We present a new temporal data model. As compared to the non-temporal counterpart, besides the valid/transaction time period defined in SQL:2011, a temporal relation under the new model has two transaction IDs. One ID corresponds to the transaction that creates the record, and the other ID corresponds to the transaction that deletes/updates the record. By introducing the transaction IDs, it is able to identify all records that are created/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 can be enhanced by taking the transaction ID as the join key instead.
- We propose a built-in temporal implementation with various optimizations, and encapsulate it into PostgreSQL [5] and Greenplum [2]. We have released our implementation publicly available.

First, our implementation completely relies on the MVCC mechanism and is lightweight. Like other temporal implementations in legacy RDBMSs, we manage current and historical records separately as well. In legacy RDBMSs, any update/delete of a current record results in synchronous migration of newly generated historical records to the historical relation. Instead, we propose an asynchronous data migration strategy, i.e., any update/delete of a current record does not cause an immediate data migration. In our design, all newly generated historical records are migrated to the historical relation 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 almost non-invasive to the originally legacy RDBMSs. Further, we utilize the KV store with various optimizations to organize historical relations, in which only data changes of historical records of the same entity are maintained, thus reducing the size of storage space.

Second, in response to the challenge that the transaction time of each record is expensive to set in legacy RDBMSs, we build an efficient transaction status manager. It maintains the status of each transaction, 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 newly generated historical record based on its transaction ID. For the current and historical records that have not yet been transferred to the historical relation, 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 the parser, query executor, and storage engine of legacy RDBMSs. Our 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 benchmarks by comparing T-SQL with Oracle, SQL Server, and MariaDB. The results show that T-SQL almost has 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 5 elaborates temporal query processing and storage management. Section 6 presents the implementation. Section 7 reports the experimental results, and Section 8 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 [16], [17], [18], [19], [20]. At this stage, the contributions mainly include temporal relation 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 types of time. The semantics of integrity constraints in the temporal data model is also enriched [12], [39]. 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 relation 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 [10], [25].

After attempts with many years to build the implementation on top of legacy RDBMSs, such as Oracle [4], DB2 [1], and Ingres [40], 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 [11], [21], [31], [44]. Although a set of temporal extensions, like TSQL2 [13], were submitted for standardization, these attempts are not successful until the adoption of SQL:2011 [25]. In response to SQL:2011, the mainstream of both commercial and open-source database management systems, including Oracle [4], IBM DB2 [1], Teradata [8], PostgreSQL [5], 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 historical relations to store current and historical data separately. ImmortalDB [28] extends SQL Server to support transaction-time queries. However, to the best of our knowledge, existing temporal RDBMSs maintain historical records associated with each attribute value, while T-SQL typically maintains the data changes of each historical record compared with its previous version. Thus, T-SQL is capable of reducing the storage overhead significantly.

Extensive efforts have been devoted to boosting the temporal query processing by proposing various data access methods. For these methods, the majority of them [22], [23], [26], [29], [30], [41], [42] are either B+-trees or R-trees based, which are particularly applicable for heap-based storage; some of them [7], [15], [36] are designed and applicable for KV stores, with the main purpose on a proper trade off among read performance, write performance, and space overhead to optimize the LSM-tree [33] or its variance. Orthogonal to the existing work, our proposed access method is still based on LSM-tree, and boosts the temporal query processing by re-organizing the key/value pairs of the historical data to skip the irrelevant key/value pairs as many as possible.

This paper is an extension of the work originally presented in [32]. Compared with [32], we optimize the storage engine by re-designing the historical data storage and refine the query processing strategy. Besides, we present the implementation of T-SQL in both centralized and decentralized RDBMSs, and release it publicly available.

3 TEMPORAL FEATURES OF T-SQL

In this section, we describe the temporal features of our system in terms of data models, 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, 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, TT is the transaction-time period, CID is the transaction ID that creates a record, and UID is the transaction ID that updates/deletes a 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 T-SQL below.

3.2.1 Creating Temporal Relations

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 the above.

3.2.2 Valid-time Queries

Valid-time queries are 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

Transaction-time queries are defined as queries over transaction relations. As compared to the regular SQL syntax, it is syntactically extended in terms of the transaction time: (1) FOR TT AS OF t, restricts records that are readable at t, (2) FOR TT FROM t_1 TO t_2, restricts records that are readable from t_1 to (but not include) t_2, (3) FOR TT BETWEEN t_1 AND t_2, restricts records that are readable from t_1 to (and include) t_2.

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., reconciliation requires a join operation on the account balance relation (R) and the expense statement relation (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 relation 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)
```

This statement will invoke a sequence of reverse operations to recover the state of records that were inserted/updated/deleted by transaction ID equal to TID. Note that the recovery fails if the other transactions' reads or writes depend on the writes of the transaction whose ID is TID. To make the system more user friendly, we report the transactions that directly depend on the transaction whose ID is TID when the recovery fails.

Example 2. *Re-consider Table 1. Suppose transaction T_i deletes Jack's account, which updates current record $r_{3.1}$ by setting $TT.ed$ to the current timestamp, and commits. Now we find T_i is an erroneous transaction, and thus rewind T_i's operations using the statement 'REWIND_TRANSACTION(T_i)'. By doing this, we recover $TT.ed$ of $r_{3.1}$ to ∞.*

3.3 Temporal Constraints

For **valid-time data model**: (1) The entity integrity is relaxed to the case that no intersection exists between the 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 relation whose valid time contains 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 logic 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 constraints 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 T-SQL. T-SQL 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 simpler hybrid non-temporal and temporal queries, and output the translated syntax tree to the query optimizer. It has two main tasks. One task is to perform the lexical and syntax analysis of input temporal queries, which follow the SQL standard defined in SQL:2011. The other task is to translate temporal qualifiers, perform semantic checks, 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 remain unchanged in the syntax tree. Note transaction-time involved operations are implemented as native support in T-SQL which will be discussed later. For illustration purposes, we give an example to show the translation of a given temporal query below.

Example 3. *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 translates the valid-time involved operations, which is underlined in the above statement, into equivalent non-temporal operations which is underlined in the following statement:

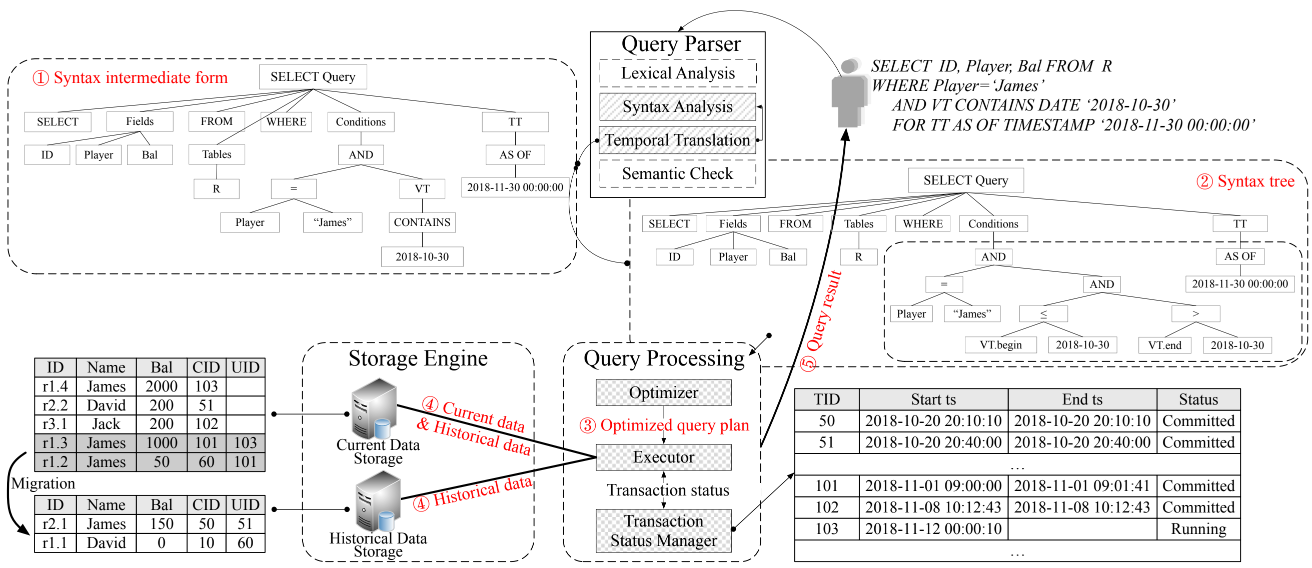
```
SELECT ID, Player, Bal FROM R
WHERE Player = 'James'
    AND VT.st ≤ DATE '2018-10-30'
    AND VT.ed > DATE '2018-10-30'
    FOR TT AS OF TIMESTAMP '2018-10-11 00:00:00'
```

For illustration purposes, we demonstrate the syntax tree and its intermediate form in Figure 1. □

• **Storage engine**. Like other temporal implementations, T-SQL also stores historical and current data separately, in which we implicitly build historical data storage to store the historical data. 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 data storage to historical data storage, 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 synchronous mode. Any update/delete of a current record produces one or multiple historical records which are then transferred from the current relation to the historical relation simultaneously. To do this, existing temporal implementations require to insert the new current record into the current relation, and migrate the newly generated historical records from the current relation to the historical relation in the same transaction.

While T-SQL processes transactional tasks on historical and current data in an asynchronous mode, we propose a hybrid data



```
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	James	150	50	51
r1.1	David	0	10	60

TID	Start ts	End ts	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 09:00:00	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
	…		

Fig. 1. System Overview

storage system encapsulated with a novel late data migration strategy, i.e., upon any update/delete of a current record does not cause an immediate data migration of the newly generated historical records. Instead, all newly generated historical records are migrated to the historical data storage when the database system starts to reclaim the storage occupied by records that are deleted or obsoleted. In T-SQL, storage reclamation 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 does not physically remove it from its relation, and (2) these deleted or obsoleted records still remain in the data pages (according to the MVCC mechanism) of the system. During the vacuum stage, we first collect all newly generated historical records from the data pages and then migrate them to the historical data storage 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 data storage during the update/delete of current records, and hence reducing the transaction latency; (2) conflicts in concurrent data access to historical data storage are completely avoided and hence result in a significant improvement of the transaction throughput for the whole system.

• **Query optimizer and executor**. The query optimizer takes a translated syntax tree as the input and generates the query execution plan for the executor. Except for the transaction-time qualifiers, the translated syntax tree is optimized just like its non-temporal counterpart by the optimizer. We provide 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 and returning the query results.

It is worth mentioning that many legacy RDBMSs, e.g., PostgreSQL, do not explicitly maintain the transaction commit timestamps associated with records. However, according to our new temporal model, each record r needs to explicitly maintain the commit time of the transaction that creates/deletes r. To address this issue, 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 set/update its transaction commit time for each of the newly generated historical records by searching it from the transaction status manager. For the current records and historical records that have not yet been transferred to the historical data storage, we are still able to obtain the transaction commit time using the same way.

Because we use current/historical data storage to store current/historical records, respectively, the executor processes the query plan over the current data storage and historical data storage separately. To fetch the current records of interest, we simply execute the query plan over the current data storage. However, it is not trivial to retrieve historical records of interest. Remind in our temporal data storage system, due to the late data migration, historical records maintained in the historical data storage are incomplete, and only querying the historical data storage could return an incomplete result. Consider that the remaining historical records are still in the current data storage, but are not visible to users. We propose an MVCC-based visibility check approach to retrieving historical records of interest from the current data storage. Details of the approach are elaborated in Section 5.3.

5 STORAGE AND QUERY PROCESSING

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

5.1 Temporal Data Storage

As discussed in the previous section, we store current data and historical data separately, and propose a late data migration strategy to transfer newly generated historical data in batch from the current data storage to the historical data storage. As our data migration relies on the conventional storage engine, we first review how the storage system works in conventional DBMSs, then present the historical data storage and its optimizations, and finally discuss the implementation of the late data migration.

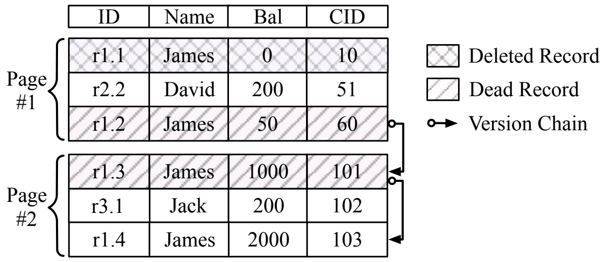


Fig. 2. An example of current data storage layout

5.1.1 Current Data Storage

Almost all popularly commercial and open-sourced RDBMSs are MVCC-based, where an update of a record r does not immediately cause a physical removal of r. Instead, r is marked as a *dead* record and a link is built from r to the new version r'. Multiple updates of records for the same entity form a chain linked from one version (record) to its next version (record).[1] Given a transaction T, any operation of T over the same entity seeks its oldest dead version via the index (if any), and follows its link iteratively to find a proper version that is visible to T based on the database isolation level [9], [27], [35]. In legacy RDBMSs, a dead record r turns to be a *deleted* record which is physically removed only if (1) r is not visible to any running or incoming transactions, and (2) a vacuum cleaner thread starts to garbage collect r. For illustration purposes, Figure 5.1.1 shows an example of how versions of the same entity are organized. Entity r_1 has four versions labeled from $r_{1.1}$ to $r_{1.4}$ stored across two data pages. $r_{1.1}$ in Page#1 is a deleted record that has been physically removed from the current data storage, and migrated to the historical data storage. $r_{1.2}, r_{1.3}$ are dead records, and a version chain is formed from $r_{1.2}$ to $r_{1.3}$ and $r_{1.3}$ to $r_{1.4}$.

For brevity, **dead records** and **deleted records** are used to denote historical records in current data storage and historical data storage, respectively.

5.1.2 Historical Data Storage

Motivated by the observation that updates of records typically limit to few attribute values, rather than every attribute value of a record, we then utilize the KV store as the historical data storage. Different from the design in the current data storage that maintains the entire record, the design in the historical data storage expects to maintain the data changes of every deleted record compared with its previous version (if any) so as to reduce the storage overhead. For this purpose, we design the historical data storage that is given in Figure 3.

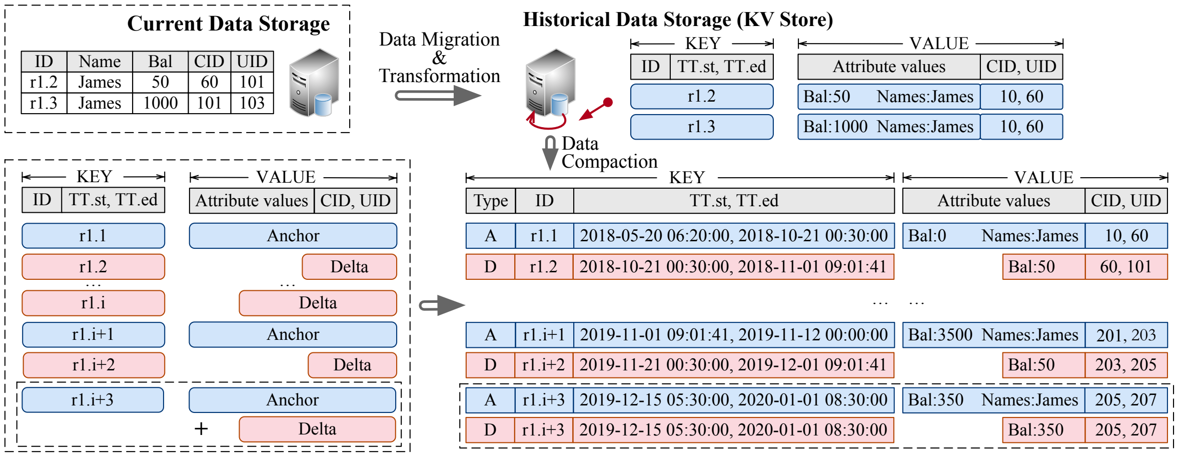
In our design, migration of records from the current data storage to the historical data storage consists of two phases: (1) data transformation (the top part of Figure 3) and (2) data compaction (the bottom part of Figure 3). Data transformation assembles dead records in the current data storage as the key-value pairs, which are then written to the KV store. Data compaction, which is periodically performed by the KV store, re-organizes the key-value pairs produced in the data transformation phase by only maintaining the data changes of every deleted record compared with its previous version (if any). It is worth mentioning that, we do not directly write the data changes of deleted records into the KV store during the migration. Instead, we split the migration into two phases. The reason is three-fold. First, data transformation is fast and hence provides a high write throughput. By doing this, it is lightweight to the original RDBMSs; second, we encapsulate the write operations into a transaction that is provided by the KV store to guarantee the fault tolerance of the migration; third, similar to VACUUM or PURGE, we do the maintenance of data changes of deleted records in batch during the data compaction phase, which is periodically performed by the KV store, and hence reduces the maintenance cost significantly.

1. In some database systems, the link is built from one version to its previous version. We emphasize that in these cases, our analysis still holds.

• **Data transformation.** As discussed before, a vacuum cleaner thread in legacy RDBMSs is invoked periodically to garbage collect dead records, wherein we migrate dead records from the current data storage to the historical data storage. We transform every dead record r to the key-value format, which is then written to the KV store. We set its key to the combination of the primary key of the corresponding dead record, $TT.st$ of r, and $TT.ed$ of r. Obviously, the key under this design is unique throughout the whole temporal relation. We set its value by assembling all remaining attribute values of r, CID of r, and UID of r. Note that in some legacy RDBMSs, like MySQL, r only consists of the data changes compared with the current record r'. In this case, we complement r with the other attribute values from r'. By doing this, we transform a dead record r to a key-value format by keeping complete information defined in the temporal model.

• **Data compaction.** Theoretically, to minimize the storage space of deleted records, for each entity, we should store every attribute value of its first deleted record, and for any subsequent update/delete, we store data changes of the corresponding deleted record compared with its previous version. In this way, however, retrieval of a certain deleted record r' incurs an expensive reconstruction cost by assembling the first deleted record with all data changes of previous versions of r'. To solve this problem, we carefully design the historical data storage by making a proper trade-off between the storage cost and query performance.

The logical layout of the KV store is shown in the bottom left of Figure 3. We maintain two kinds of deleted records, namely *anchor* record, and *delta* record. An anchor record maintains every attribute value of a deleted record, and a delta record only maintains data changes of a deleted record compared with its previous version. Retrieval of a certain deleted record r' invokes a seek of its most recent anchor record r, and re-construct r' by assembling r' with all data changes of versions from r to r'. To make a good trade-off between the storage cost and the query performance, we build an anchor record either periodically or reaching a certain number of delta records. Surprisingly, building a delta record from the data transformation for each entity is not trivial because the data changes of a delta record need to capture the difference between the data transformation and its previous version in terms of an anchor record. To solve this problem, for the latest version of each entity, we maintain both the anchor record and delta record. Upon an append of a deleted record r, we fetch the last anchor record r' of the entity, compute the data changes of r to r', write one anchor record of r with the other delta record of the data changes to the KV store, and delete r'. To differentiate anchor records with delta records in the KV store, as well as boost the temporal query performance which is discussed in Section 5.3, we concatenate one-bit character prefix, with ‘A’ as the anchor record and ‘D’ as the delta record, to the key, shown in the bottom right of Figure 3. Take an example shown in Figure 3. $r_{1.1}$ is the first history record of entity r_1 and $r_{1.1}$ is set as an anchor record. Suppose we create an anchor record for every i delta records. Thus, the subsequent records from $r_{1.2}$


Current Data Storage
Data Migration & Transformation
Historical Data Storage (KV Store)
Data Compaction
KEY
VALUE

ID	Name	Bal	CID	UID
r1.2	James	50	60	101
r1.3	James	1000	101	103

ID	TT.st, TT.ed	Attribute values	CID, UID
r1.2		Bal:50 Names:James	10, 60
r1.3		Bal:1000 Names:James	10, 60

ID	TT.st, TT.ed	Attribute values	CID, UID
r1.1		Anchor	
r1.2		Delta	
…		…	
r1.i		Delta	
r1.i+1		Anchor	
r1.i+2		Delta	
r1.i+3		Anchor + Delta	

Type	ID	TT.st, TT.ed	Attribute values	CID, UID
A	r1.1	2018-05-20 06:20:00, 2018-10-21 00:30:00	Bal:0 Names:James	10, 60
D	r1.2	2018-10-21 00:30:00, 2018-11-01 09:01:41	Bal:50	60, 101
		… …		
A	r1.i+1	2019-11-01 09:01:41, 2019-11-12 00:00:00	Bal:3500 Names:James	201, 203
D	r1.i+2	2019-11-21 00:30:00, 2019-12-01 09:01:41	Bal:50	203, 205
A	r1.i+3	2019-12-15 05:30:00, 2020-01-01 08:30:00	Bal:350 Names:James	205, 207
D	r1.i+3	2019-12-15 05:30:00, 2020-01-01 08:30:00	Bal:350	205, 207

Fig. 3. Logical layout of the KV store

to $r_{1.i}$ are all delta records, while $r_{1.i+1}$ is created as an anchor record. $r_{1.i+3}$ is the last deleted record, and thus, we create both an anchor record and a delta record for it. It is worth mentioning that, by introducing the prefix, every anchor record in the KV store is physically ordered before all delta records.

• **Implementation.** We migrate dead records from the current data storage to the historical data storage during VACUUM or PURGE which is periodically invoked by legacy RDBMSs. To provide automatic data migration, we modify the logic of the vacuum by introducing `Migrate()` function. Our implementation is said to be lightweight and almost non-invasive in that we do not introduce new modules and simply copy the dead records, which are supposed to be physically removed by the vacuum cleaner, to the historical data storage. Besides, since the vacuum is invoked periodically, we do data migration in a late and asynchronous way.

Algorithm 1 gives the details about function `Migrate()`. We use variable $Set(\mathcal{P})$ to represent a set of pages that contain dead records to be migrated as the input and num to store the number of migrated records as the output (line 1). Each page in $Set(\mathcal{P})$ containing dead records is sequentially checked (line 3). In order to guarantee fault tolerance, for each page, we use a transaction to ensure that the migration of records in a page can be atomic and durable (line 4,11). For any dead record r to be vacuumed in $\mathcal{P}$, r is transformed to a key-value pair (denoted as $pair$) by the function `encode2kv()` (line 7), written to RocksDB, and deleted from the current data storage (line 3-11).

Almost all key-value stores are LSM-tree based [33], where the data compaction is periodically invoked to combine the same key-value pairs from different files. We re-organize the key-value pairs as anchor records and delta records during the data compaction phase of the KV store by introducing a new function, namely `Compress()`. In our implementation, we use RocksDB [37] as the KV store. Algorithm 2 shows the pseudocode of `Compress()`. We use the variable it as the input, which is the iterator pointing to the key-value pairs that are generated in the data transformation but have not yet been re-organized as anchor or delta records. To guarantee atomicity, we encapsulate `Compress()` function into a transaction (denoted as $\mathcal{T}$) provided by RocksDB (line 2). We seek the last anchor record it_a of the same entity that shares the same key $it \rightarrow key$ (line 4-5), and copy it_a to kv_a. Note that key-value pairs from the same entity are continuously stored in it. We obtain the number of delta records of the entity since the most recent anchor record except it_a using the

Algorithm 1: Data Migration
```
1  Function Migrate ($Set(\mathcal{P})$):
       input  : $Set(\mathcal{P})$, a set of pages which contain records
                to be vacuumed;
       output: $num$, the number of migrated records;
2      $num \leftarrow 0$;
3      for $\mathcal{P}$ in $Set(\mathcal{P})$ do
4          KV_store::start ($\mathcal{T}$);
5          for $r \in \mathcal{P}$ do
6              if toVacuum ($r$) then
7                  $pair$ = encode2kv ($r$);
8                  KV_store::put ($\mathcal{T}$, $pair$);
9                  physically delete $r$ from $\mathcal{P}$;
10                 $num$ ++;
11         KV_store::commit ($\mathcal{T}$);
12     return $num$;
```

function `getNumOfDeltas` (line 6). In the real application, this value could be associated with the last anchor record, i.e., it_a, of the entity. We then delete the last anchor record it_a (line 7-8) and iteratively examine each key-value pair of the same entity (line 9-18) to generate either delta record (line 10-12) or anchor record (line 13-14) by comparing the number of currently generated delta records with a pre-defined threshold $\mathcal{I}$. For the last key-value pair of an entity, we generate both its anchor record (line 18-19), and delta record (line 10-12), and update the number of delta records (line 20). Finally, after all key-value pairs pointed by it are processed, transaction $\mathcal{T}$ commits (line 21).

As shown in Figure 3, records $r_{1.2}$ and $r_{1.3}$ are transferred into the key-value format and migrated to historical data storage by calling `Migration()`. Later, an asynchronous data compaction is invoked, which re-organizes $r_{1.2}$ as an anchor record and creates an anchor record and a delta record for $r_{1.3}$.

5.2 Transaction Time Maintenance

Setting the transaction-time for records in current data storage is not trivial. Recall that in the temporal data model, 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

Algorithm 2: Data Compaction

```
1  Function Compress(it):
       input : $it$, the iterator points to the first record need
               to be re-organized;
2      KV_store::start($\mathcal{T}$);
3      while $it$ do
4          $it_a \leftarrow$ KV_store::seekprev($it \rightarrow key$);
5          $kv_a \leftarrow$ copy($it_a \rightarrow kv$); $key \leftarrow (it_a \rightarrow key)$;
6          $num_d \leftarrow$ KV_store::getNumOfDeltas($it_a$);
7          if $it_a \rightarrow kv \wedge num_d > 0 \wedge it_a.prev()$ then
8              KV_store::delete($it_a \rightarrow kv$);
9          do
10             if mod($num_d, \mathcal{I}$) then
11                 KV_store::put($\mathcal{T}$,toDelta($it \rightarrow kv$));
12                 $num_d$ ++;
13             else
14                 KV_store::put($\mathcal{T}$,toAnchor($it \rightarrow kv$));
15             $kv_a \leftarrow$ copy($it \rightarrow kv$);
16             $it \leftarrow it.next()$;
17         while $it \wedge key = it \rightarrow key$;
18         if mod($num_d, \mathcal{I}$) then
19             KV_store::put($\mathcal{T}$,toAnchor($it \rightarrow kv$));
20         KV_store::setNumOfDeltas($it, num_d$);
21     KV_store::commit($\mathcal{T}$);
```

transaction $\mathcal{T}_e$, its $TT.ed$ should be set to the commit time of $\mathcal{T}_e$. To enable fast query processing, legacy RDBMSs do not maintain the commit time of the transaction that creates/updates/deletes with each record. In practice, it is inefficient to set the transaction time for all inserted/deleted/updated records upon a transaction is committed because they may not be in memory. Interestingly, SQL:2011 leaves the transaction time up to SQL-implementations to pick up an appropriate value. While many temporal implementations in legacy RDBMSs, either pick up the start time of a transaction or the time of a operation that inserts/updates/deletes the record, we argue that this could potentially cause an incorrect result based on the database isolation levels.

For these reasons, we propose a transaction status manager that maintains status for transactions. For each transaction $\mathcal{T}$, we maintain a transaction log including the commit time and the transaction status (running/committed/aborted). 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 besides the status of $\mathcal{T}$ upon $\mathcal{T}$ is committed/aborted. In our design, since the size of each transaction log is fixed, we can efficiently retrieve the log of a transaction using its ID. For example, the log of a transaction with $ID = 10$ is the 10^{th} log in Page#1. By doing this, we can fetch the transaction commit time on the fly by searching the corresponding transaction status from the transaction manager during either the data migration or the temporal query processing. For example, we set the transaction time $TT.ed$ of a transforming record r to the commit time of the transaction that updates/deletes r by searching it from the transaction status manager.

5.3 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 focus on the transaction-time query processing with the objective to efficiently retrieve current and historical records of interest, given that they are stored separately. Unless otherwise specified, a relation mentioned in this section is referred to as a transaction-time relation.

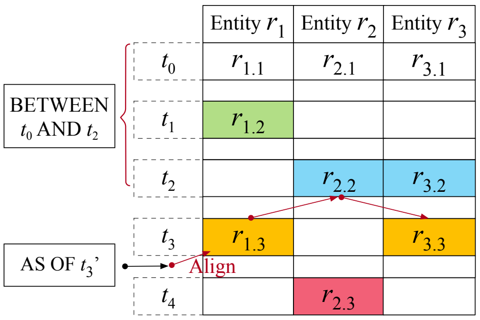


Fig. 4. Transaction-time query processing

Correctly answering transaction-time queries requires first reconstructing snapshot(s) according to a given time point or time interval, then retrieving all records that are visible in this/these snapshot(s). Temporal data that satisfy the requirements specified in the query conditions are finally returned. As defined in Section 3, transaction-time queries are divided into three categories: (1) FOR TT AS OF time t, (2) FOR TT FROM t_1 TO t_2, and (3) FOR TT BETWEEN t_1 AND t_2. For ease of illustration, we refer to the first category as *time-travel query*, the second and the third as *time-slice query*. For illustration purposes, we give an example below to show how a transaction-time query works.

Example 4. *Suppose there exist three entities, r_1, r_2 and r_3 shown in Figure 5.3. 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_4 updates $r_{2.2}$ to $r_{2.3}$. Suppose we issue a time-travel query at time t'_3, as discussed above, we need to reconstruct the snapshot at time t_3. We then align all records that are visible in this snapshot, i.e., $r_{1.3}, r_{2.2}, r_{3.3}$, as the query result. If we issue a time-slice query between t_0 and t_2, $r_{1.1}, r_{2.1}, r_{3.1}, r_{1.2}, r_{2.2}, r_{3.2}$ are returned since they are visible in one of the snapshots reconstructed at time t_0, t_1 and t_2 respectively.* □

As observed from Example 4, a transaction-time query may require to retrieve both current records and historical records from the database system. Specifically, retrieval of current records follows the same logic like that in legacy RDBMSs, while retrieval of historical records needs to be a careful design. For historical records, on one hand, dead records that have been deleted/updated but not yet been migrated to the historical data storage, are still in the current data storage. For these dead records, we need to retrieve them from the current data storage; on the other hand, for the deleted records that have been migrated from current data storage to historical data storage, we retrieve them simply from the historical data storage. Note that we need to carefully consider the retrieval order. If we first retrieve deleted records in the historical data storage and then dead records in the concurrent data storage, some records that should be examined are lost due to the concurrent data migration. To solve this problem, we make a reverse order by first retrieving dead records in the current data storage, and then deleted records in the historical data storage. By doing

Algorithm 3: Retrieving dead records

```
1  Function DeadRecRead($P$,$\mathcal{C}$):
       input : $P$, a data page; $\mathcal{C}$, temporal condition;
       output: $Set(rec)$, the result set;
2      $rec \leftarrow$ the record in $\mathcal{P}$ from which we start to scan;
3      while $rec$ do
4          if !SnapshotCheck($rec$) then
5              $rec \leftarrow next(rec)$;
6              continue;
7          if TemporalCheck($rec.TT$,$\mathcal{C}$) then
8              $Set(rec) \leftarrow Set(rec) \cup \{rec\}$;
9          $rec \leftarrow next(rec)$;
10     return $Set(rec)$;
```

Algorithm 4: Retrieving deleted records

```
1  Function DeletedRecRead($Set(\mathcal{K})$,$\mathcal{C}$):
       input : $Set(\mathcal{K})$, a set of primary keys;
               $\mathcal{C}$, temporal condition;
       output: $Set(rec)$, the result set;
2      if $Set(\mathcal{K})$=$NULL$ then
           // Check the secondary index
3          for $idx \in index$ do
4              if TemporalCheck($idx.value$,$\mathcal{C}$) then
5                  $Set(\mathcal{K}) \leftarrow Set(\mathcal{K}) \cup \{idx.key\}$;
6      for $\mathcal{K}$ in $Set(\mathcal{K})$ do
7          $it_a \leftarrow$ KV_store::seekprev($\mathcal{K}$,$\mathcal{C}$);
8          $kv_a, kv_d \leftarrow$ copy($it_a \rightarrow kv$);
9          do
10             if TemporalCheck($kv_a.TT$,$\mathcal{C}$) then
11                 $Set(rec) \leftarrow Set(rec) \cup \{kv_a\}$;
12                 if $\mathcal{C}.type$=$as\_of$ then break;
13             if $it_d \leftarrow$KV_store::seeknext($kv_a.key$)
                then SearchDelta($kv_d$,$it_d$,$\mathcal{C}$,$Set(rec)$);
14         while $it_a \leftarrow it_a.next() \wedge kv_a \leftarrow$copy($it_a \rightarrow kv$);
15     return $Set(rec)$;
16 Function SearchDelta($Set(\mathcal{K})$,$\mathcal{C}$):
17     $num \leftarrow 0$;
18     do
19         $num$++;
20         $kv_d \leftarrow$ combine($kv_d$, $it_d \rightarrow kv$);
21         if TemporalCheck($kv_d.TT$,$\mathcal{C}$) then
22             $Set(rec) \leftarrow Set(rec) \cup \{kv_d\}$;
23             if $\mathcal{C}.type$=$as\_of$ then break;
24     while $it_d \leftarrow it_d.next() \wedge num \leq \mathcal{I}$;
```

this, we can guarantee the result are complete. Although some results could be duplicated examined in both current data storage and historical data storage, this does not affect the correctness and we can issue a late de-duplication to solve this problem.

• **Retrieval of dead records.** Remind in MVCC-based RDBMSs, the query executor invokes either sequential scan or index scan to find records of interest. To do a sequential scan over a relation R, either every current record or dead record of R is examined. To identify dead records of interest, we additionally examine every dead record r whether or not it is visible to given temporal constraints $\mathcal{C}$ based on Equation 1–3. Consider a temporal constraint $\mathcal{C}$ in a transaction-time query. $\mathcal{C}$ maintains the instance time t, denoted as $\mathcal{C}.t$, for time-travel queries and the time interval t_1, t_2, denoted as $\mathcal{C}.t_1, \mathcal{C}.t_2$, respectively, for time-slice queries.

Category 1: FOR TT AS OF time t,

$$r.TT.st \leq \mathcal{C}.t \wedge r.TT.ed > \mathcal{C}.t \tag{1}$$

Category 2: FOR TT FROM t_1 TO t_2,

$$r.TT.st < \mathcal{C}.t_2 \wedge r.TT.ed > \mathcal{C}.t_1 \tag{2}$$

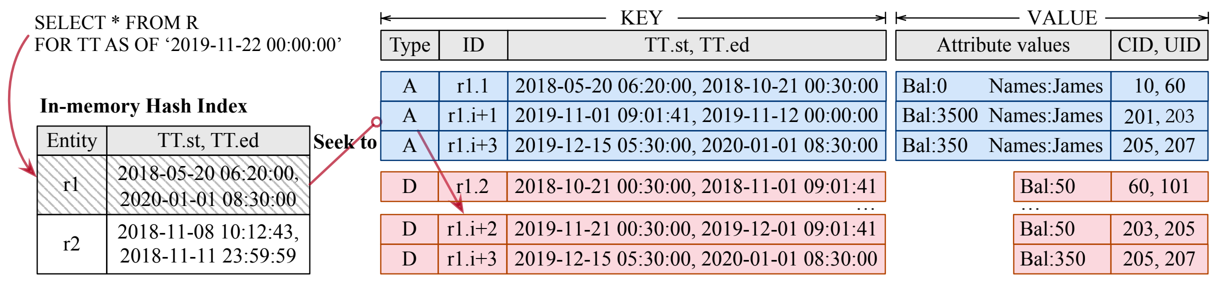
Category 3: FOR TT BETWEEN t_1 AND t_2,

$$r.TT.st \leq \mathcal{C}.t_2 \wedge r.TT.ed > \mathcal{C}.t_1 \tag{3}$$

To do an index scan, the first version of each entity r is identified, and the other dead records of r are sequentially examined by following the version chain of the first version. Similarly, whether r is visible to a given temporal constraints $\mathcal{C}$ is examined based on Equation 1–3. We develop function `DeadRecRead()` to fetch proper dead records and Algorithm 3 shows the pseudocode. By taking a data page $\mathcal{P}$ and temporal constraint $\mathcal{C}$ as the input, we sequentially check records in $\mathcal{P}$ (line 2). We start to scan from rec, which is either the first record in $\mathcal{P}$ or the record pointed by the index. We check whether a record is visible to the current transaction's snapshot (line 4–6) and satisfies the given temporal constraints (line 7–8). If either of them is violated, the next record is fetched by function $next()$ (line 5,9) and continues to check on it. In sequential scan, function $next()$ returns the next record on this page while in index scan it returns the next record according to the version chain. We introduce function `TemporalCheck()` to examine whether a record rec satisfies the temporal constraints based on Equation 1–3.

• **Retrieval of deleted records.** To retrieve a deleted record of interest, our basic idea is to invoke a seek of its most recent anchor record r, and assemble r' with all data changes of versions from r to r'. Thanks to this special design, queries with primary keys specified in query conditions can be processed efficiently. However, in many cases, if there is no primary key but the time interval specified in the query condition, the above retrieval of deleted records cannot work properly. To address this issue, we build a lightweight in-memory hash index, shown on the left of Figure 5. In this index, we set the index key to the primary key of an entity. We set the index value to the interval for each entity from $TT.st$ of the first anchor record to $TT.ed$ of the last anchor record. By traversing the index, we can obtain whether an entity has any historical record that satisfies the temporal constraint by checking its index value with the time interval specified in the query condition. If an entity is verified as a candidate, we then follow the same logic of that in transaction-time queries with primary keys specified to retrieve deleted records.

Algorithm 4 shows how to retrieve deleted records. In function `DeletedRecRead()`, for queries without primary keys specified, we scan the secondary index to find all qualified entities and put primary keys of them into $Set(\mathcal{K})$ (line 2–5). For each entity (line 6), we seek to the nearest anchor record it_a according to $\mathcal{K}$ and temporal conditions $\mathcal{C}$ (line 7–8). We examine kv_a (line 10–12) and iteratively get all remaining anchor records as well as records that have not been compressed (line 14). We then seek to the following delta record (if any) according to kv_a and iteratively



Entity	TT.st, TT.ed
r1	2018-05-20 06:20:00, 2020-01-01 08:30:00
r2	2018-11-08 10:12:43, 2018-11-11 23:59:59

Type	ID	TT.st, TT.ed	Attribute values	CID, UID
A	r1.1	2018-05-20 06:20:00, 2018-10-21 00:30:00	Bal:0 Names:James	10, 60
A	r1.i+1	2019-11-01 09:01:41, 2019-11-12 00:00:00	Bal:3500 Names:James	201, 203
A	r1.i+3	2019-12-15 05:30:00, 2020-01-01 08:30:00	Bal:350 Names:James	205, 207
D	r1.2	2018-10-21 00:30:00, 2018-11-01 09:01:41	Bal:50	60, 101
		...	...	
D	r1.i+2	2019-11-21 00:30:00, 2019-12-01 09:01:41	Bal:50	203, 205
D	r1.i+3	2019-12-15 05:30:00, 2020-01-01 08:30:00	Bal:350	205, 207

Fig. 5. Retrieval of deleted records

fetch delta records of interest through function `SearchDelta()` (line 10). In function `SearchDelta()`, kv_d is assembled with a delta record $it_d \leftarrow kv$ during iteration by calling function `combine()` (line 20) and every kv_d that satisfies $\mathcal{C}$ is copied into $Set(rec)$ (line 22). Finally, the result set $Set(rec)$ returns (line 15). We shall give an example to illustrate Algorithm 4. As shown in Figure 5, an AS_OF query retrieves all deleted records that are visible to the snapshot at timestamp ts='2019-11-22 00:00:00'. To do this, we first search the in-memory hash index by examining each entity with ts locating in interval [$TT.st$, $TT.ed$), and in this case, we find the qualified entity is r_1. We further seek to the closest anchor record $r_{1.j}$ of entity r_1 in the historical data storage where $TT.st$ of $r_{1.j}$ is less than ts and $TT.st$ of $r_{1.j'}$, the next anchor record of $r_{1.j}$, is greater than ts. If ts is located in the interval [$TT.st$, $TT.ed$) of $r_{1.j}$, we return $r_{1.j}$; otherwise, we further seek to the first delta record $r_{1.m}$ with its $TT.st$ is greater than $TT.ed$ of $r_{1.j}$, and make a sequential scan from $r_{1.m}$ until $r_{1.n}$ with ts locating in its [$TT.st$, $TT.ed$). Note that we return the result by assembling $r_{1.n}$ with $r_{1.j}$ and all previous delta records of $r_{1.n}$. In our example, we find $r_{1.i+2}$ is the proper record and the result is assembled by $r_{1.i+2}$ and $r_{1.i+1}$.

5.4 Discussion

In this section, we theoretically analyze the effect of interval $\mathcal{I}$ on the space consumption of T-SQL. Note that $\mathcal{I}$ is the number of delta records between every two adjacent anchor records of the same entity.

For ease of illustration, let H be the collection of historical records, M be the total number of entities in H, r_i be an entity in H. Further, we use $\mathcal{C}(H)$, and $\mathcal{C}(r_i)$ to denote the space consumption of H, and r_i, respectively. Besides, we use $\mathcal{C}(r_i^a)$, and $\mathcal{C}(r_i^d)$ to denote the total size of anchor records and delta records of r_i, respectively. Hence, for $i \in \mathbb{N}^+$, we have:

$$\mathcal{C}(H) = \sum_{i=1}^{M} \mathcal{C}(r_i) = \sum_{i=1}^{M} [\mathcal{C}(r_i^a) + \mathcal{C}(r_i^d)]. \tag{4}$$

For an entity r_i, we denote its total number of historical records as N_i. Generally, for an entity r_i, we denote the average size of an anchor record and a delta record as $\mathcal{S}(r_i^a)$ and $\mathcal{S}(r_i^d)$, respectively. Hence, we have:

$$\mathcal{C}(r_i^a) = \lceil \frac{N_i}{\mathcal{I}} \rceil \times \mathcal{S}(r_i^a), \mathcal{C}(r_i^d) = N_i - \lfloor \frac{N_i}{\mathcal{I}} \rfloor) \times \mathcal{S}(r_i^d) \tag{5}$$

In Equation 5, $\lceil \frac{N_i}{\mathcal{I}} \rceil = \lfloor \frac{N_i}{\mathcal{I}} \rfloor$ holds when $N_i \gg \mathcal{I}$. Based on Equation 4 and 5, we then have:

$$\mathcal{C}(r_i) = [\frac{\mathcal{S}(r_i^a) - \mathcal{S}(r_i^d)}{\mathcal{I}} + \mathcal{S}(r_i^d)] \times N_i \tag{6}$$

Because $\mathcal{S}(r_i^a)$ is always equivalent to or greater than $\mathcal{S}(r_i^d)$, we can calculate the derivatives as:

$$\begin{aligned} \frac{\partial \mathcal{C}(r_i)}{\partial \mathcal{I}} &= -\frac{[\mathcal{S}(r_i^a) - \mathcal{S}(r_i^d)] \times N_i}{\mathcal{I}^2} \le 0, \\ \frac{\partial^2 \mathcal{C}(r_i)}{\partial^2 \mathcal{I}} &= \frac{2 \times [\mathcal{S}(r_i^a) - \mathcal{S}(r_i^d)] \times N_i}{\mathcal{I}^3} \ge 0 \end{aligned} \tag{7}$$

Based on Equation 7, we conclude that for an entity r_i, it has positive gains on the storage consumption by increasing $\mathcal{I}$, but the gains diminish. Based on Equation 4, we argue that this conclusion still holds for H. We also verify this conclusion in Section 7.1.

6 IMPLEMENTATION

In this section, we present the implementation of T-SQL in PostgreSQL and Greenplum, which are open-sourced centralized and distributed RDBMSs, respectively. We release our implementation publicly available via https://github.com/dbiir/T-SQL.

6.1 Implementation in PostgreSQL

To enable temporal support in PostgreSQL, T-SQL extends the query engine (including parser, query optimizer, executor), and add a component called transaction status manager.

• **Parser.** We first extend the parser to make it capable of recognizing temporal qualifiers in function `pg_parse_query()`. Second, in function `pg_rewrite_query()`, we update the syntax tree produced by the parser by translating valid-time operators into equivalent non-temporal operators, and keeping transaction-time operators unchanged.

• **Query optimizer and executor.** We extend the query optimizer and executor to make them capable of processing transaction-time operators. If there exist transaction-time operators in the translated syntax tree, the query executor searches temporal data of interest over both the current data storage and the historical data storage. To do this, in function `ExecSeqScan()` and `ExecIndexScan()`, besides the search of current records and dead records of interest (Algorithm 3) from the current data storage, we add function `DeletedRecRead()` to fetch deleted records of interest from the KV store (Algorithm 4).

• **Storage engine.** We use the native heap-based storage in PostgreSQL as the current data storage to maintain current records and dead records. In PostgreSQL, every current record or dead record contains all attribute values. We integrate RocksDB into PostgreSQL to maintain deleted records. To do this, in function `PostmasterMain()`, we additionally start a RocksDB process when the PostgreSQL instance starts. To interchange data between RocksDB and the query executor, we employ the shared memory. When processing a query, the query executor puts specific KV operations into the shared memory. Worker threads in RocksDB then fetch operations from the shared memory, execute these operations and give back results through the shared memory to the query executor. We integrate the late data migration into function `heap_prune_chain()` where the vacuum is pruning a data page that contains dead records. Besides, we encapsulate our proposed data compaction logic into function `BackgroundCompaction()` in RocksDB.

• **Transaction status manager.** We build a transaction status manager based on CLOG, which is also called commit log in PostgreSQL, to maintain transaction information. For each transaction $\mathcal{T}$, besides the transaction status (running/committed/aborted), we additionally maintain the commit time of $\mathcal{T}$ in CLOG. The main modifications are in the source code file `clog.c`.

6.2 The Extension to GreenPlum

Greenplum is a distributed database system mainly for analytics and technically built on top of PostgreSQL. There are one master and several segments in a Greenplum cluster, each of which is a PostgreSQL instance. The master is responsible for query dispatching while the segments execute corresponding sub-queries since data is partitioned and separated in every segment. To enable temporal support in Greenplum, the main issue is to extend the implementation in PostgreSQL into a distributed manner. To do this, T-SQL first extends the parser (in the master) and the storage engine (in the segments) as discussed in Section 6.1. Then T-SQL additionally adds a component called timestamp oracle [34], [38], extends the query executor to be capable of processing distributed temporal queries, and enables transaction status managers to record the status of distributed transactions.

In order to uniformly assign timestamps for transactions, we employ the timestamp oracle to assign timestamps in a strictly increasing order. We implement the timestamp oracle as a raft-based cluster to ensure high availability and provide a robust timestamp allocation service. The timestamp oracle is called by the master to request timestamps for transactions. A temporal query is decomposed into several sub-queries in the master and executed in corresponding segments individually. In each segment, the implementation of sub-query execution is according to Section 6.1. The master is responsible for coordinating distributed transactions. Every transaction contacts the timestamp oracle once during the commit phase to set the commit timestamp. We then introduce a transaction status manager into the master and every segment to maintain the status of distributed transactions. For a distributed transaction, its transaction logs are stored in the master and all participant segments. To do this, we modify the query executor of the master in function `CdbDispatchDtxProtocolCommand()` to transfer the commit timestamp of the current transaction to corresponding segments when dispatching commit commands. Subsequently, the commit timestamp of a transaction is stored in its logs. In this way, a given transaction's commit time can be fetched directly from the local transaction status manager and without additional network communication. We also release our Greenplum-based implementation in our repository.

7 EVALUATION

Our experiments are carried out on a server with an Intel 24-core Xeon 2.4GHz CPU, 128GB Memory, and 480GB SSDs, running a CentOS 7.4 operation system with kernel version 3.10.106. We compare T-SQL, which enables temporal support in PostgreSQL, with Oracle 11g, SQL Server 2017, and MariaDB 10.3.11. We use the default configurations for the temporal database systems, except that the buffer pool and the thread pool are set to 8GB and 48, respectively if the systems provide these two parameters.

TABLE 2
Temporal queries for YCSB benchmark

Query	Statement
Q1	SELECT * FROM R FOR TT AS OF t WHERE ID=id
Q2	SELECT * FROM R FOR TT FROM t_1 TO t_2 WHERE ID=id
Q3	SELECT * FROM usertable FOR TT AS OF t
Q4	SELECT * FROM usertable FOR TT FROM t_1 TO t_2

We conduct the experiments using the following one real and three widely-used synthetic benchmarks. We use YCSB benchmark to quantitatively evaluate the trade off between the storage space and temporal query performance. Besides, we use Tencent-RB to study the efficiency of T-SQL in a real application and TPC-series benchmarks to compare T-SQL with other common RDBMSs by introducing built-in temporal support.

• **YCSB** [14] is a synthetic benchmark modeling Yahoo! Cloud Services, simulating large-scale Internet applications. It contains a single table with a primary key and 10 other columns. We create a YCSB table with 10 million records, each of which occupies 1KB. By default, we set the total number of transactions to 500,000 for each experiment run, and calculate the throughput per second based on the total execution time for these transactions. We use default settings provided by the official YCSB tool with a default workload, i.e., workloada (50% reads + 50% writes).

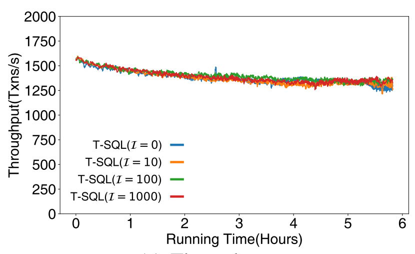
• **Tencent-RB** is a real benchmark abstracted from the Tencent billing service platform. It has two relations, in which one relation R depicts user account balances which are fairly stable with nearly 500 million of records while the other relation W depicts the expense statements. We collect one month (30 days) data and run a query to examine the account balance with the expense statement for each user per day.

• **TPC-C** [43] 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 by the number of new-order transactions per minute. To support temporal data management, we add transaction time as an additional attribute to each of 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.

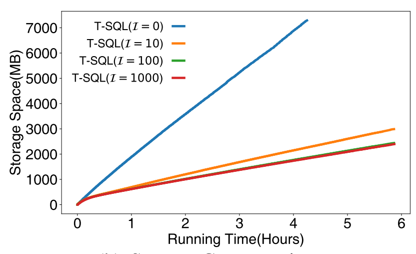
• **TPC-BiH** [24] is a recently proposed benchmark, which is particularly designed for the evaluation over temporal databases. It builds on the solid foundations of the TPC-H benchmark but extends it with a rich set of temporal queries and update scenarios that simulate real-life temporal applications from SAP's customer.

7.1 Evaluation on YCSB

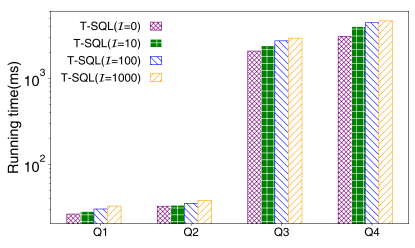
We study the trade off between the storage consumption and temporal query performance using the YCSB benchmark. We first study the storage consumption by varying the interval (denoted as $\mathcal{I}$), i.e., the number of delta records between every two anchor records, from 0 to 1000. We run the workloada provided by YCSB for 6 hours with 4 client threads and plot the throughput and storage consumption in Figure 6(a) and 6(b) respectively. As we can see, by introducing the delta records that maintain data changes only, the storage consumption can be reduced by 2.2×



(a) Throughput

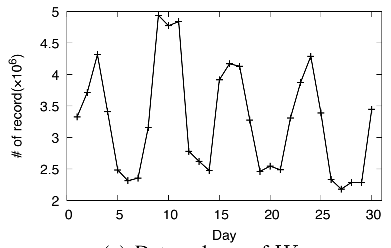


(b) Storage Consumption

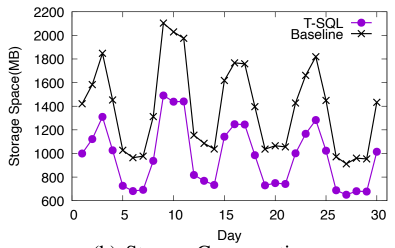


(c) Query Performance

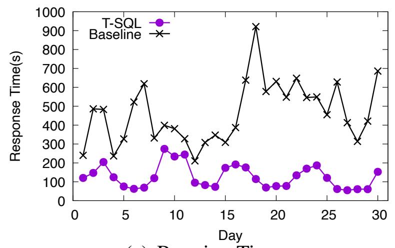
Fig. 6. Evaluation on YCSB



(a) Data volume of W

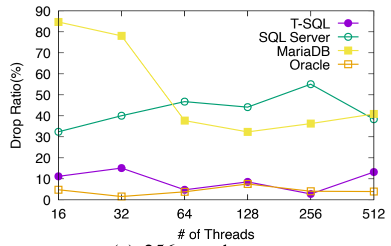


(b) Storage Consumption

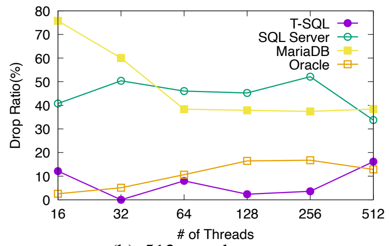


(c) Running Time

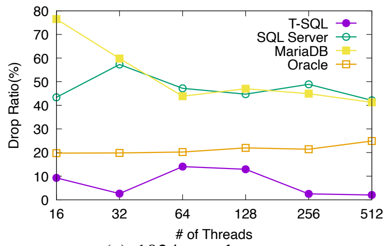
Fig. 7. Evaluation on Tencent-RB



(a) 256 warehouses



(b) 512 warehouses



(c) 1024 warehouses

Fig. 8. Effect on the drop ratio by introducing temporal support

when $\mathcal{I} = 10$ and $2.9\times$ when $\mathcal{I} = 100$, respectively. We also observe that the storage consumption is roughly coincident when $\mathcal{I}$ is set to 100 and 1000, respectively. We confirm that although the storage consumption is reduced by increasing $\mathcal{I}$, there is an obvious trend of diminishing return.

We further study the temporal query performance by varying the interval $\mathcal{I}$. After we run the YCSB workloada for 1 hours, we invoke four typical temporal queries that are shown in Table 2 and plot the running time in Figure 6(c). Note that t, t_1, and t_2 denote the timestamp by running the YCSB workloada for 30 minutes, 15minutes, and 45minutes, respectively. We observe that the larger $\mathcal{I}$ takes, the longer running time consumes. This verifies that the cost of assembling a historical record is positively related to $\mathcal{I}$. By properly leveraging the performance and the storage consumption, we set $\mathcal{I} = 100$ in the remaining experiments.

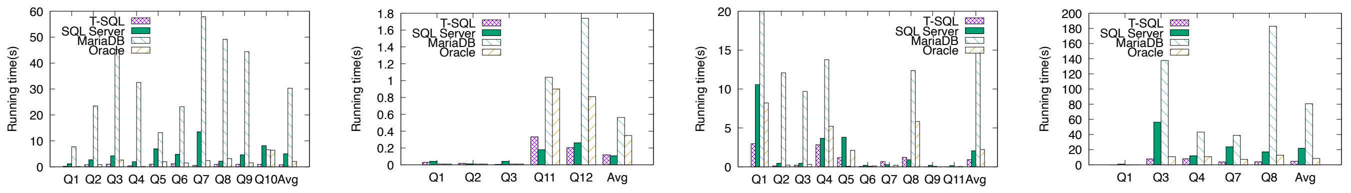
7.2 Evaluation on Tencent-RB

We firstly run a query to do daily account reconciliation on the baseline and collect the query performance. As a comparison, we transform the regular query into an equivalent temporal query (as presented in Section 3), run it on T-SQL, and collect the query performance as well. Figure 7(c) shows the result by comparing the performance of baseline and T-SQL. It can be observed that T-SQL is in general significantly faster than the baseline, ranging from $1.4\times$ to $9.3\times$, depending on the data size. In this one month duration, the performance of T-SQL is less sensitive when the data size varies, and is always better than the baseline. The main reason is the transformed temporal query eliminates the join of unnecessary data. The baseline executes 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 7(a). As T-SQL has already maintained the changed accounts automatically, of which the number varies from 2 million to 5 million, T-SQL only needs to join the changed accounts with W, which results in much less join computation than that of the baseline. Again, we can observe a great degradation of storage consumption from Figure 7(b).

7.3 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 8 shows the effect on the drop ratios by varying the number of data warehouses. From the figures, we make two observations. First, the drop ratio of T-SQL is 7% on average, and varies from 2% to 16%, showing that T-SQL's temporal implementation is lightweight. Second, the drop ratio of T-SQL 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, T-SQL shows its superiority, achieving 4% to 21% smaller drop ratio than Oracle. In all cases, the drop ratio of T-SQL is significantly smaller than that of MariaDB and SQL Server, further showing its lightweight feature. As repeatedly discussed before, T-SQL transfers the historical data in batch only during the



(a) Time-travel Query (b) Pure-key Queries (c) Bi-temporal Query (d) Range-timeslice Query

Fig. 9. Performance study on TPC-BiH

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.

7.4 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 over TPC-BiH.

We report the running time in Figures 9(a)˜(d). For answering time-travel queries, we observe that on average T-SQL runs $2\times$, $5\times$, $30\times$ faster than Oracle, SQL Server, MariaDB, respectively. Interestingly, for answering pure-key queries, SQL Server performs slightly better than T-SQL, followed by Oracle, and MariaDB. Note that answering queries Q11 and Q12 is the bottleneck to process the pure-key queries. We observe that scanning the Custom relation is the dominant cost of answering query Q11 and Q12. Because the Custom relation is seldom updated, scanning its current records is much more expensive than scanning its historical records. The reason why SQL Server performs slightly better than T-SQL is that it makes various optimizations over the retrieval of the current records. We argue that this paper focuses on the efficiency of retrieving historical records, such optimizations are orthogonal to our work. 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 KV store, with a secondary index built on the transaction time.

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

8 Conclusion

In this paper, we present T-SQL, a lightweight yet efficient built-in temporal implementation in database systems. 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 YCSB and TPC-series benchmarks, and the results show T-SQL almost has minimal performance loss by introducing the temporal features, and performs the best for most of the temporal queries.

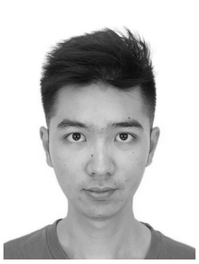
Acknowledgments

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 61972403, 61732014 and U1711261, and Tencent Rhino-Bird Joint Research Program. Wei Lu and Xiaoyong Du are the corresponding authors.

References

[1] DB2. https://www.ibm.com/analytics/us/en/db2.
[2] Greenplum. https://greenplum.org/.
[3] Mariadb. https://mariadb.com/kb/en/library/system-versioned-tables/.
[4] Oracle. https://www.oracle.com.
[5] PostgreSQL. https://www.postgresql.org.
[6] SQL Server. https://docs.microsoft.com/en-us/sql/relational-databases/tables/temporal-tables.
[7] J. Ahn, M. A. Qader, W. Kang, H. Nguyen, G. Zhang, and S. Ben-Romdhane. Jungle: Towards dynamically adjustable key-value store by combining lsm-tree and copy-on-write b+-tree. In *HotStorage*. USENIX Association, 2019.
[8] 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.
[9] 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.
[10] M. Böhlen and C. Jensen. *Temporal Data Model and Query Language Concepts*, pages 437–453. 12 2003.
[11] C. X. Chen and C. Zaniolo. Sqlst: A spatio-temporal data model and query language. In *ER*, volume 1920 of *Lecture Notes in Computer Science*, pages 96–111. Springer, 2000.
[12] J. Chomicki. Efficient checking of temporal integrity constraints using bounded history encoding. *ACM Trans. Database Syst.*, 20(2):149–186, 1995.
[13] 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.
[14] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, pages 143–154. ACM, 2010.
[15] N. Dayan and S. Idreos. Dostoevsky: Better space-time trade-offs for lsm-tree based key-value stores via adaptive removal of superfluous merging. In *SIGMOD Conference*, pages 505–520. ACM, 2018.
[16] 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.
[17] S. K. Gadia and C. Yeung. A generalized model for a relational temporal database. In *SIGMOD Conference*, pages 251–259. ACM Press, 1988.
[18] 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.
[19] 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.
[20] 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.
[21] C. S. Jensen and R. T. Snodgrass. Temporal data management. *IEEE Trans. Knowl. Data Eng.*, 11(1):36–44, 1999.

[22] 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.
[23] 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.
[24] 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.
[25] K. G. Kulkarni and J. Michels. Temporal features in SQL: 2011. *SIGMOD Record*, 41(3):34–43, 2012.
[26] 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.
[27] 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*, 22(6):2633–2653, 2019.
[28] 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.
[29] D. B. Lomet, M. Hong, R. V. Nehme, and R. Zhang. Transaction time indexing with version compression. *Proc. VLDB Endow.*, 1(1):870–881, 2008.
[30] D. B. Lomet and F. Li. Improving transaction-time DBMS performance and functionality. In *ICDE*, pages 581–591. IEEE Computer Society, 2009.
[31] N. A. Lorentzos and Y. G. Mitsopoulos. SQL extension for interval data. *IEEE Trans. Knowl. Data Eng.*, 9(3):480–499, 1997.
[32] W. Lu, Z. Zhao, X. Wang, H. Li, Z. Zhang, Z. Shui, S. Ye, A. Pan, and X. Du. A lightweight and efficient temporal database management system in TDSQL. *Proc. VLDB Endow.*, 12(12):2035–2046, 2019.
[33] P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.
[34] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *OSDI*, pages 251–264. USENIX Association, 2010.
[35] D. R. K. Ports and K. Grittner. Serializable snapshot isolation in postgresql. *PVLDB*, 5(12):1850–1861, 2012.
[36] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *SOSP*, pages 497–514. ACM, 2017.
[37] RocksDB. https://rocksdb.org/.
[38] O. Shacham, Y. Gottesman, A. Bergman, E. Bortnikov, E. Hillel, and I. Keidar. Taking omid to the clouds: Fast, scalable transactions for real-time cloud analytics. *Proc. VLDB Endow.*, 11(12):1795–1808, 2018.
[39] A. P. Sistla and O. Wolfson. Temporal conditions and integrity constraints in active database systems. In *SIGMOD Conference*, pages 269–280. ACM Press, 1995.
[40] M. Stonebraker, E. Wong, P. Kreps, and G. Held. The design and implementation of INGRES. *ACM Trans. Database Syst.*, 1(3):189–222, 1976.
[41] 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.
[42] 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.
[43] TPC-C. http://www.tpc.org/tpcc/.
[44] J. R. R. Viqueira and N. A. Lorentzos. SQL extension for spatio-temporal data. *VLDB J.*, 16(2):179–200, 2007.



Zhanhao Zhao is currently a PhD student at the School of Information and the Key Lab of Data Engineering and Knowledge Engineering, Renmin University of China. His research interests include distributed database systems and transaction processing.

Wei Lu is currently an associate professor at Renmin University of China. He received his Ph.D degree in computer science from Renmin University of China in 2011. His research interests include query processing in the context of spatiotemporal, cloud database systems and applications.

Hongyao Zhao is currently a PhD student at the School of Information and the Key Lab of Data Engineering and Knowledge Engineering, Renmin University of China. His research interests include distributed databases and transaction processing.

Zongyan He is currently an undergraduate student at the School of Information, Renmin University of China. His research interests include cloud computing and big data.

Haixiang Li is currently a senior expert at Tencent. His research interests include transaction processing, query optimization, distributed consistency, high availability, database system architecture, cloud database and distributed database systems.

Anqun Pan is a technical director of Tencent Billing Platform Department, has more than 15 years of experience in the research and development of distributed computing and storage systems. He is currently responsible for the research and development of distributed database system (TDSQL).

Xiaoyong Du is a professor at Renmin University of China. He received his Ph.D. degree from Nagoya Institute of Technology in 1997. His research focuses on intelligent information retrieval, high performance database and unstructured data management.