Implementation and Evaluation of a Partitioned Store for Transaction-Time Databases

Manigantan Sethuraman

December 10, 2003

TR-76

A TIMECENTER Technical Report

Title	Implementation and Evaluation of a Partitioned Store for Transaction-Time Databases Copyright © 2003 Manigantan Sethuraman. All rights reserved.
Author(s)	Manigantan Sethuraman
Publication History	December 2003, A TIMECENTER Technical Report

Abstract

There has been an increasing need for transaction-time databases. Such databases provide the ability to maintain the history of the data as it evolves. Along with this ability comes the problem of reduction in performance of current queries, over an increasing amount of data. Temporally partitioning the store of a transaction-time database provides a promising solution to this problem. We implemented a temporally partitioned store in the transaction-time database τBerkeleyDB and evaluated its effect on a wide range of queries and update transactions. We also propose changes to MySQL language to provide transaction-time support and have modified τMySQL to support nonsequenced select and a temporally partitioned store.

Contents

1 Introduction

Databases are used to store the data available in the real world, so as to make the data easily accessible and more manageable. Conventional databases keep track of the data as it changes in the real world, but they do not maintain the history of the data as it evolves. Today, there are many applications that have a need for tracking the evolution of data objects. Some of these applications use this information to predict the future of the data objects, while others use this information to perform application specific tasks. To take care of this need *temporal database systems* were introduced [9].

A temporal database system keeps track of the history of the data as the data changes. The time information that characterizes the point at which the data changed can be classified as *valid time*, *transaction time* and *user-defined time* [8]. Valid time is the time when the data changes in the real world, transaction time is the time when the transaction to record the change in data executes in the database, and user-defined time is simply a value entered by the user without further interpretation by the database system. Conventional database systems already support user-defined time, but the other two types of time are not supported by them.

This thesis focuses on *transaction-time databases*. A transaction-time database records the history of a database's transactions rather than just the result of the final transaction. For example, in a system maintaining employee information, if a person's salary is changed from 5000 to 6000, a conventional database would keep track of the fact that the person's salary is 6000 now, whereas a transaction-time database will keep track of the fact that the person's salary was initially 5000 and was changed to 6000, along with the information on when this change occurred. The change history maintained by a transaction-time database cannot be changed, which ensures its reliability. The reliability of the change history information is important to applications like auditing and billing, since they use it to discover anomalous activities in the system. Although the change history of the data is important, it is not this content that is frequently accessed, but it is the current version of the data that is more frequently used by applications. This calls for better access to the current version of the data than the history versions of the data.

Using the conventional access methods to access current version of the data in a transaction-time database will provide a response which is slower than that provided by a conventional database storing just the current version. So working forward based on the idea of providing better access to the current version, its attractive to have a *temporally partitioned store* [3]. A temporally partitioned store divides data into separate storage areas based on the time attribute. For example, in a system maintaining employee information, the current information of all employees will be kept in one store and all the history information will be kept in another. This is expected to provide better access to the current version of the data, since the current information has been separated from the history information and is maintained separately. As a part of this thesis we implement a temporally partitioned store in the available transaction-time database and evaluate the effect of temporally partitioned indices on a wide range of queries and update transactions.

The available transaction-time database is τ*BerkeleyDB* [1]. It has been built by adding transaction-time support to BerkeleyDB. In order to expose the features in τBerkeleyDB through an SQL interface, MySQL has been chosen as the SQL processing engine. In order to provide transaction-time support through MySQL, MySQL and its language need to be modified so that it can understand temporal queries and interact with τBerkeleyDB accordingly. This thesis proposes the changes for the MySQL language to support transaction-time. The changes have been proposed with reference to the changes proposed in “Database Language SQL — Part 7: Temporal” [4], the change proposals “Adding Valid Time to SQL/Temporal” [11] and “Adding Transaction Time to SQL/Temporal” [10] and the addendum [7]. τMySQL is MySQL with transaction-time support. We have implemented support for *nonsequenced select* in τMySQL. A nonsequenced select allows the user to query the version history of the data and look at the changes it has gone through. We have also implemented support for temporal partitioning in τMySQL. This allows the τMySQL user to specify whether the transaction-time database being created needs to be temporally partitioned.

The rest of this thesis is organized as follows. We review the related work in Section 2. In Section 3 we describe the design decisions taken by us. In Section 4 we propose the changes to be done to τBerkeleyDB interface to support the creation and manipulation of databases with a temporally partitioned store. In Section 5 we propose the changes to be done to MySQL language for providing transaction-time support and also discuss the applicability of these changes to MySQL. In Section 6 we explain architectural changes done by us. Section 7 provides the implementation details of the modifications performed by us. In Section 8 we present the experimental results showing the effect of the partitioned store on the existing system. Section 9 discusses possible future work.

2 Related Work

A temporally partitioned store has two storage areas, the *current store* and the *archival store* [3]. The current store contains current versions which can satisfy all current queries and the archival store holds the history versions. A query is called *current* if it involves only current data and does not concern data that has been updated or logically deleted.

The archival store can be organized so that the cost of accessing the store can be reduced significantly. Various ways to organize the archival store are *reverse chaining*, *accession lists*, *clustering*, *stacking* and *cellular chaining* [3]. Reverse chaining and accession lists maintain pointers to the various versions of a particular key for providing better access, whereas stacking, clustering and cellular chaining work towards the same goal by placing versions of a particular key close to each other. Further improvement in performance of queries can be obtained by indexing, which is also based on the idea of maintaining pointers to the records for quicker access. The index can also be partitioned in order to have an index for the current store and another for the archival store. Indexing and cellular chaining are generic forms of the pointer based methods and clustering based methods respectively. The data in BerkeleyDB is clustered by key. So, we chose to implement a temporally partitioned index so that we can evaluate the effect of applying both indexing and clustering to the system, as this has not been evaluated by previous experiments. Ahn and Snodgrass also suggest that more exhaustive evaluation of the system should be done with a wider range of queries, which is also a part of this thesis.

Salzberg and Tsotras [2] have provided an exhaustive comparison of access methods for temporal databases. A major part of their paper discusses transaction-time databases. They compare the performance of various access methods based on three types of queries, viz. *transaction pure-timeslice query*, *transaction pure-key query* and *transaction range-timeslice query*. A transaction pure-timeslice query is one which looks for all objects that are alive at a given time. A transaction pure-key query is one which looks for all object versions that have a given key value. A transaction range-timeslice query is one which looks for all objects that are alive at given time and whose key values fall in a given range. They classify access methods as *key-only methods*, *time-only methods* and *time-key methods*. Key-only methods work towards clustering all versions of a given key. Time-only methods cluster versions belonging to a given time interval. Time-key methods cluster data by both transaction-time and key. The temporally partitioned index implemented by us uses the existing key-only access method available in BerkeleyDB.

There have also been many papers on the various ways to implement temporal support in a conventional database. The design goals for such an implementation are to achieve *upward compatibility* with minimal coding effort, to achieve *temporal upward compatibility*, and to reuse the conventional database's modules to the maximum extent possible [5, 6]. Upward compatibility is required in order to avoid changes to code accessing the conventional database. Temporal upward compatibility makes it possible to convert an existing application accessing a non-temporal database into one accessing a temporal database, without affecting the application. Our goal has been to follow the above mentioned design goals as closely as possible.

"Database Language SQL — Part 7: Temporal" [4] defines the changes that would have to be applied to an SQL implementation in order to support temporal data. The change proposals "Adding Valid Time to SQL/Temporal" [11], "Adding Transaction Time to SQL/Temporal" [10] and the addendum [7] propose the changes that would have to be applied to SQL/Temporal in order to provide valid-time and transaction-time support respectively. We have analyzed the applicability of these changes to MySQL language.

3 Design Decisions

The following sections explain the various design decisions taken during the implementation of the partitioned store for τBerkeleyDB and the preparation of the change proposal for MySQL language.

3.1 Partitioned Store for τBerkeleyDB

As discussed earlier we have implemented a temporally partitioned store in the transaction-time database τBerkeleyDB. τBerkeleyDB uses a B-Tree based access method for indexing its data. In order to temporally partition the index we decided to have two B-Tree structures, one that indexes the current data and the other that indexes the archive data. Since the current store's size will be equivalent in size to the data store that a conventional database would have maintained, the response of the new system to conventional queries is expected to be equivalent to that provided by the conventional database.

A temporally partitioned store and index will affect the performance of queries as well as insert, update and delete operations. In a case where only the current version is kept in the current store and all the archive versions are moved to the archive store immediately, the current query performance is expected to be high, but the insert, update and delete operations will face the overhead of pushing the archive version to the archive store. We could keep the current record and a few archive records in the current store and later push the archive records to the archive store when they cross a threshold. We decided to keep only current records in the current store, so as to provide the maximum benefit to current queries.

In order to have least impact on the existing modules of τBerkeleyDB and other systems using τBerkeleyDB, we decided to implement the partitioned store with minimal modifications to τBerkeleyDB interface. We also decided to encapsulate all the modifications into one layer and abstract it in such a way that the other layers will not be aware of the presence or absence of the partitioned store. In order to provide flexibility, we chose to make the store available or unavailable based on the presence or absence of an input flag.

3.2 Change Proposal for MySQL Language

The changes to MySQL have been proposed with the following objectives.

- Minimize changes to MySQL language.
- Maximize consistency with respect to MySQL language.
- Maximize consistency with respect to the proposed period data type in SQL/Temporal [4].
- Maximize consistency with respect to the SQL/Temporal valid-time and transaction-time change proposals [11, 10, 7].
- Maintain temporal upward compatibility.

In order to minimize the impact on MySQL it was decided not to add the period data type to MySQL. Instead the `datetime` data type available in MySQL has been used to achieve the required functionality.

4 Proposed Changes to τBerkeleyDB

τBerkeleyDB allows the creation and manipulation of transaction-time databases. We propose the following change to its interface to support the creation and manipulation of databases with a temporally partitioned store.

- The `DB->open` function opens the database represented by the `file` and `database` arguments for both reading and writing. The `type` argument is of type `DBTYPE`, and must be set to `DB_BTREE` for transaction-time databases. The `flags` and `mode` arguments specify how files will be opened and/or created if they do not already exist. The signature of the interface is shown below.

  ```
  int DB->open(DB *db, DB_TXN *txnid, const char *file,
   const char *database, DBTYPE type, u_int32_t flags, int mode);
  ```

  A new flag called `DB_PARTITION` shall be used. The flag can be specified by setting a specific bit in the `flags` argument. When creating a temporal database (that is, with the `DB_TEMPORAL` flag specified), if this flag is specified, a temporal database is created with a partitioned store. When opening an existing temporal database with a partitioned store, specifying this flag or not makes no difference; the database will always be opened as a temporal one with a partitioned store. When opening an existing regular database or a temporal database without a partitioned store this flag must not be specified, otherwise an error will be returned.

- A new function `is_partitioned` shall be provided with the following signature.

```
int is_partitioned(DB *db);
```

The function shall take a pointer to a database as the argument and return 1 if that database has a partitioned store, and 0 otherwise.

- A new function `DB->set_partitioning` shall be provided with the following signature.

  ```
  int DB->set_partitioning(DB *db, u_int32_t flag);
  ```

  This function shall partition the store of the given database if the value of the argument `flag` is 1 and the given database is not already partitioned. If the value of the argument `flag` is 0 and the given database has a partitioned store, then the current and archive store would be merged into one, to undo the partitioning. If the given database does not support transaction-time or if the arguments specified do not satisfy any one of the above cases then an error will be returned.

- A new function `DB->set_transactiontime` shall be provided with the following signature.

  ```
  int DB->set_transactiontime(DB *db, u_int32_t flag);
  ```

  This function shall convert the given database into a transaction-time database if the value of argument `flag` is 1 and the given database is not a transaction-time database. If the value of the argument `flag` is 0 and the given database is a transaction-time database with or without a partitioned store, then the database will be converted to a non transaction-time database. If the arguments specified do not satisfy any one of the above cases then an error will be returned.

The other functions in the τBerkeleyDB interface shall not have any changes to their signatures. The data retrieval functions like `DB->get` and `DBC->c_get`, on being invoked with a transaction-time database as input, return the current record in the database with respect to the current system time. This behavior shall be changed so that the functions return the current record with respect to the current timestamp of the environment. The current timestamp of the environment is the current system time by default, but it can be set to a different value using the `DB_ENV->set_tx_timestamp` function.

```
int DB_ENV->set_tx_timestamp(DB_ENV *dbenv, time_t *timestamp);
```

This function is already available in BerkeleyDB and allows the user to recover the system to the time specified by the timestamp rather than to the most current possible date. Changing the current timestamp to affect the behavior of data retrieval functions should not affect the recovery of the system, since recovery occurs only when the database is being opened.

5 Proposed Changes to MySQL Language

The following sections propose changes to MySQL Language to provide transaction-time and partitioned store support.

5.1 Proposed Changes to SQL-92

The syntax for the changes proposed in this section is given as extensions to ANSI SQL-92. The applicability of these changes to MySQL is discussed in the prose surrounded by a box.

5.1.1 Clause 6 Scalar Expressions

5.1.1.1 Subclause 6.8 <datetime value function>

Function

Specify a function yielding a value of type datetime.

Format

1) In the Format, add the following new alternative to <current date value function>:

| CURRENT_DATE <left paren> TRANSACTIONTIME <right paren>

2) In the Format, add the following new alternative to <current time value function>:

| CURRENT_TIME <left paren> TRANSACTIONTIME <right paren>

3) In the Format, add the following new alternative to <current timestamp function>:

| CURRENT_TIMESTAMP <left paren> TRANSACTIONTIME <right paren>

Syntax Rules

No additional Syntax Rules.

Access Rules

No additional Access Rules.

General Rules

1. (Insert this GR) If TRANSACTIONTIME is specified in the <datetime value function>s CURRENT_DATE, CURRENT_TIME and CURRENT_TIMESTAMP, then these functions shall return the current date, current time, and current timestamp respectively, consistent with the transaction time of the transaction in which this reference is contained.

Note to proposal reader: The addition of the reserved word TRANSACTIONTIME has been proposed in Section 5.2.3.1.

Applicability to MySQL

This change is directly applicable to MySQL since the format followed by MySQL is similar to the SQL-92 format and MySQL already supports the functions CURRENT_DATE, CURRENT_TIME and CURRENT_TIMESTAMP. The function CURRENT_TIMESTAMP(TRANSACTIONTIME) shall provide a precision of second in MySQL 3.23.49 (as that version does not support a precision of microsecond) and a precision of microsecond in MySQL 4.1.1(as that version supports a precision of microsecond).

5.1.2 Clause 16 Session Management

1) Insert the following Subclause, "<set timestamp statement>" to SQL-92 immediately following Subclause 16.5, "<set local time zone statement>".

5.1.3 Subclause 16.6 “<set timestamp statement>”

Function

Set the current timestamp for the current SQL-session.

Format

```
<set timestamp statement> ::=
        SET TIMESTAMP <set timestamp value>

<set timestamp value> ::=
            <datetime value expression>
        |   DEFAULT
```

Syntax Rules

1. (Insert this SR) The data type of <datetime value expression> shall be TIMESTAMP.

Access Rules

No additional Access Rules.

General Rules

1. (Insert this GR) Case:

   a) If DEFAULT is specified, then the current timestamp of the SQL-session is set to the current timestamp of the system with time zone displacement equivalent to the current time zone displacement of the SQL-session.

   b) Otherwise, the current timestamp of the SQL-session is set to the value of <datetime value expression>.

Applicability to MySQL

This feature is available in MySQL in the following format.

```
<set timestamp statement> ::=
        SET TIMESTAMP <equals operator> <set timestamp value>

<set timestamp value> ::=
            <unsigned integer>
        |   DEFAULT
```

The <unsigned integer> contained in <set timestamp value> represents the Unix epoch timestamp value, to be set as the current timestamp. Currently, setting a timestamp value for an SQL-session only alters the value returned by the <datetime value function>s. This needs to be enhanced such that the result of queries on transaction-time tables are evaluated with respect to the current timestamp of the SQL-session. The update and insert statements shall continue to use the current system timestamp rather than the current timestamp of the SQL-session.

5.2 Proposed Changes to SQL/Temporal

The syntax for the changes proposed in this section is given as extensions to “Database Language SQL — Part 7: Temporal” [4] with reference to the changes proposed by the previous change proposals [11, 10, 7]. In following subsections, we first give the change proposal as verbatim from the transaction-time change proposal [10] and the addendum [7], then apply it to MySQL in prose surrounded by a box.

5.2.1 Clause 3 Definitions, Notations, and Conventions

5.2.1.1 Subclause 3.1 Definitions

1) Add the following terms.

g) **row with transaction-time support**: A row with transaction-time support is a row with an associated transaction time, which is a value of a period data type, with elements of the transaction-time precision, which is implementation-defined.

h) **transaction time of a row with transaction-time support**: The transaction time of a row with transaction-time support is the period P such that BEGIN(P) denotes the time at which the row was inserted and END(P) denotes the time when the row was updated or (logically) deleted.

i) **table has transaction-time support**: A table with transaction-time support is one in which each row is a row with transaction-time support.

j) **transaction-time value of a table with transaction-time support at a transaction time**: The transaction-time value of a table with transaction-time support, TT, at a specified time, T, is the table without transaction-time support comprising rows with identical values for the fields of the rows of TT associated with transaction times that overlap T.

k) **current transaction-time value of a table with transaction-time support**: The current transaction-time value of a table with transaction-time support is the transaction-time value of that table at transaction time CURRENT_TIMESTAMP.

Applicability to MySQL

The definitions would apply as they are to MySQL, except that there would be two fields STARTTIME and STOPTIME of `datetime` data type denoting the transaction time instead of the one field of period data type. The two fields will contain the time when the row was inserted or changed and the time when the row was logically removed (that is, the transaction-time of the transaction that modified or deleted the row), respectively. For example, considering that the employee record of Franziska was inserted on 1995-01-01 and updated on 1995-02-01, the employee record of Lilian was created on 1995-01-01 and deleted on 1995-03-02 and the the employee record of Therese was inserted on 1995-02-01, the STARTTIME and STOPTIME for these records shall be maintained as follows.

ename	salary	STARTTIME	STOPTIME
Franziska	5000	1995-01-01	1995-02-01
Franziska	6000	1995-02-01	9999-12-31
Therese	4000	1995-02-01	9999-12-31
Lilian	4500	1995-02-02	1995-03-02

The transaction-time precision is dependent on the precision of `datetime` data type. In MySQL 3.23.49 the precision of the `datetime` data type is seconds.

5.2.2 Clause 4 Concepts

1) Insert the following Subclause, “Tables”, to SQL/Temporal immediately following Subclause 4.2.3, “Period predicates”.

5.2.2.1 Subclause 4.3 Tables

Every table descriptor also includes:

– An indication of whether the table has transaction-time support or does not have transaction-time support.

1) Insert the following Subclause, “Integrity constraints”, to SQL/Temporal immediately following Subclause 4.3, “Tables”.

5.2.2.2 Subclause 4.4 Integrity Constraints

Every constraint descriptor also includes:

– An indication of whether the constraint is specified without TRANSACTIONTIME, with TRANSACTIONTIME but without NONSEQUENCED, or with NONSEQUENCED TRANSACTIONTIME.

– The transaction-time period, if any, associated with the constraint.

1) Insert the following Subclause, “Meaning of statements on tables with transaction-time support”, to SQL/Temporal immediately following Subclause 4.4, “Integrity constraints”.

5.2.2.3 Subclass 4.5 Meaning of statements on tables with transaction-time support

The meaning of queries on tables with transaction-time support is parallel to and orthogonal with those queries on tables with valid-time support. The concepts of temporal upward compatibility, sequenced transaction, and nonsequenced transaction apply consistently to queries, integrity constraints, assertions, views, and cursors.

Modifications on tables with transaction-time support are always performed on the current transaction-time value of the table, with the resulting rows of the new value having a transaction-time period P such that BEGIN(P) is CURRENT_TIMESTAMP, thereby ensuring the append-only nature of transaction-time support. For updates and deletions, the ending bound of the transaction times of the rows that are affected are set to the value of CURRENT_TIMESTAMP. For rows that have not been updated or deleted, the ending bound is always CURRENT_TIMESTAMP.

Applicability to MySQL

The changes proposed in the Clause 4, “Concepts” are directly applicable to MySQL. CURRENT_TIMESTAMP is available in MySQL as a function that returns the current system time.

5.2.3 Clause 5 Lexical Elements

5.2.3.1 Subclause 5.1 <token> and <separator>

1. In the Format, add the following two new alternatives to <reserved word>:

```
| TRANSACTIONTIME
| NONSEQUENCED
```

Applicability to MySQL

This change is directly applicable to MySQL, as the keywords to be added do not conflict with the existing keywords in MySQL. In addition to this change, the following action needs to performed.

- In the Format, add the following new alternatives to <non-reserved word>:

```
| PARTITIONED
```

5.2.4 Clause 6 Scalar Expressions

1) Insert the following two Subclauses, “<item reference>” and “<table reference>”, to SQL/Temporal immediately preceding Subclause 6.2, “<set function specification>”.

5.2.4.1 Subclause 6.1 <item reference>

Function

Reference a column, parameter, or variable.

Format

No additional Format items.

Syntax Rules

1. (Replace SR4) If IR does not contain an <item qualifier>, then

   Case:

   a) If IR is contained within the scope of one or more exposed <table or query name>s, <correlation name>s, or <routine>s whose associated tables or <parameter list>s include a column or parameter whose <identifier> is IN, then

      i) Let the phrase possible qualifiers denote those exposed <table or query name>s, <correlation name>s, and <routine name>s.

      ii) Case:

         1) If the most local scope contains exactly one possible qualifier, then the qualifier IQ equivalent to that unique exposed <table or query name>, <correlation name>, or <routine name> is implicit.

         2) If there is more than one possible qualifier with the most local scope, then:

            a) Each possible qualifier shall be a <table or query name> or a <correlation name> of a <table reference> that is directly contained in a <joined table> JT.

            b) CN shall be a common column name in JT.

            c) The implicit qualifier IQ is implementation-dependent. The scope of IQ is that which IQ would have had if JT had been replaced by the <table reference>:
            (JT) AS IQ

      iii) Let V be the table or parameter list associated with IQ.

   b) If IR is contained in a <value expression> of a <time option> that is simply contained in a <query expression> QE, then

      i) The implicit qualifier IQ is implementation-dependent. The scope of IQ is that which IQ would have had if the <query expression body> QEB of QE had been replaced by the <table reference>:
      (QEB) AS IQ

      ii) Let V be the table associated with IQ.

   Note to proposal reader: The original SR4 appears as Case a. This adds Case b.

Access Rules

No additional Access Rules.

General Rules

No additional General Rules.

Applicability to MySQL

This change, which specifies the scope of an <item reference> contained in a <time option>, is directly applicable to MySQL.

5.2.4.2 Subclause 6.2 <table reference>

Function

Reference a table.

Format

No additional Format items.

Syntax Rules

1. (Replace SR2a) If a <table reference> TR is contained in a <from clause> FC with no intervening <derived table>, then the scope clause SC of TR is the <select statement: single row> SS or innermost <query specification> that contains FC. The scope of the exposed <correlation name> or exposed <table or query name> of TR is the <select list>, <from clause>, <where clause>, <group by clause>, and <having clause> of SC, together with the <join condition> of all <joined table>s contained in SC that contain TR and the <time option> of SS.

   Note to proposal reader: This adds "and the <time option> of SS" to the scope.

Access Rules

No additional Access Rules.

General Rules

No additional General Rules.

Applicability to MySQL

This change, which specifies the scope of a <table reference>, is directly applicable to MySQL.

5.2.4.3 Subclause 6.5 <period value expression>

1) In the Format, add the following new alternative to <period primary>:

```
    | <transactiontime function>
```

2) In the Format, add the following two BNF productions:

```
<transactiontime function> ::=
        TRANSACTIONTIME <left paren> <transactiontime argument> <right paren>

<transactiontime argument> ::=
              <item qualifier>
        |     <value expression>
```

3) Insert the following two Syntax Rules:

1. (Insert this SR) The data type of <transactiontime function> shall be <period type>, with an element precision of the transaction-time precision.
2. (Insert this SR) The <value expression> of a <transactiontime function> shall be of row R. If R does not have transaction-time support, then it shall have a field named TRANSACTIONTIME of a period data type, with an element precision of the transaction-time precision.

Note to proposal reader: The precision of a table with transaction-time support was specified in Subclause 3.1, "Definitions" as implementation-defined.

4) Insert the following General Rules:

1. (Insert this GR) Case:
   a) If <transactiontime argument> is <item qualifier>, then let R be the row for which <transactiontime function> is evaluated.
   b) If <transactiontime argument> is <value expression>, then let R be the resulting row.
2. (Insert this GR) Case:
   a) If R has transaction-time support, then the value of the <transactiontime function> is the transaction-time period of R.
   b) If R does not have transaction-time support, then the value of the <transactiontime function> is the value of the field of R named TRANSACTIONTIME.
3. (Insert this GR) Let the value of the <transactiontime function> be T. If LAST(T) is the end of time, then replace LAST(T) with CURRENT_TIMESTAMP in the transaction-time precision.

Note to proposal reader: The end of time was specified in Subclause 3.1, "Definitions".

Applicability to MySQL

This change can be applied to MySQL by performing the following actions.

- Add <transactiontime function> as an alternative to <value expression> specified in Subclause 6.4.
- In the Format, replace <transactiontime function> BNF production by the following.

```
<transactiontime function> ::=
                BEGIN <left paren> TRANSACTIONTIME <left paren>
                <transactiontime argument> <right paren> <right paren>
            |   END <left paren> TRANSACTIONTIME <left paren>
                <transactiontime argument> <right paren> <right paren>
```

  The keywords BEGIN and END are already available in MySQL as alternatives to the <non-reserved word> BNF production.

- Replace the Syntax Rule 1 above with the following rule. The data type of <transactiontime function> shall be <datetime type>, with an element precision of transaction-time precision.
- In the Syntax Rules and General Rules instances that expect a field named TRANSACTIONTIME of period data type, shall now expect two fields of `datetime` data type named STARTTIME and STOPTIME. The value of the <transactiontime function> is the value of the field named STARTTIME if the production containing BEGIN is used. Similarly, the value of the <transactiontime function> is the value of the field named STOPTIME if the production containing END is used.

5.2.5 Clause 7 Query Expressions

Insert the following Subclause, "<query expression>", to SQL/Temporal immediately following Subclause 7.3, "<period value constructor>".

5.2.5.1 Subclause 7.4 <query expression>

Function

Specify a table.

Format

1) In the Format, replace the <query expression> BNF production with:

```
<query expression> ::=
        [ <with clause> ] <temporal query expression body>
```

2) Add the following BNF productions:

```
<temporal query expression body> ::=
        [ <time option> ] <query expression body>

<time option> ::=
        <transactiontime option>

<transactiontime option> ::=
            [ NONSEQUENCED ] TRANSACTIONTIME [ <value expression> ]
```

Syntax Rules

1. (Insert this SR) The data type of <value expression> of <transactiontime option> shall be <period type>, with an element precision of the transaction-time precision.

   NOTE 6 - Subclause 6.3, "<item reference>" restricts the scope of column names in <value expression>.

2. (Insert this SR) If TRANSACTIONTIME is specified and NONSEQUENCED is not specified in the <transactiontime option> that is contained in the <time option> that is simply contained in <query expression> QE, then for each explicit or implicit <item qualifier> IQ with a scope clause of QE or of a <query expression> that is contained in QE without an intervening <from clause>, IQ shall be associated with a table with transaction-time support.

   Note to proposal reader: This ensures that sequenced transaction queries are only evaluated "over" tables with transaction-time support.

3. (Insert this SR) If TRANSACTIONTIME is specified in the <time option> of a <query expression> Q, then either Q shall be simply contained in a <from clause> or Q shall be the outermost <query expression>.

4. (Insert this SR) If NONSEQUENCED is specified in a <transactiontime option> TO that is contained in <time option>, then TO shall not contain a <value expression>.

5. (Insert this SR) Let T be the result of the <query expression>.

   Case:

   a) If TRANSACTIONTIME is specified and NONSEQUENCED is not specified in the <transactiontime option> that is contained in <time option>, then T shall be a table with transaction-time support.

   b) Otherwise, T shall be a table without transaction-time support.

Access Rules

No additional Access Rules.

General Rules

1. (Insert this GR) Case:

   a) If TRANSACTIONTIME is specified and NONSEQUENCED is not specified in the <transactiontime option> that is contained in <time option>, then the result of <temporal query expression body> TQEB during each transaction time granule T of the transaction-time precision is the result of the <query expression body> of TQEB with each leaf generally underlying table with transaction-time support with no intervening <from clause> replaced with its state at transaction time T. If <value expression> VE is specified in the <transactiontime option> that is contained in <time option>, then for each row R resulting from the initial result, that is, before the following replacements, of TQEB,

      Case:

      i) If the value of VE and the transaction-time period VP of R overlap, then the resulting transaction-time period of R is the result of
      (VE P_INTERSECT VP).

      ii) Otherwise, R is not included in the final result of TQEB.

   b) If NONSEQUENCED TRANSACTIONTIME is specified in <time option>, then the result of <temporal query expression body> TQEB is the result of the <query expression body> of TQEB with each leaf generally underlying table with transaction-time support with no intervening <from clause> QTT replaced by the table without transaction-time support comprising rows with identical values for the fields of the rows of QTT. The descriptor of that table is the same as the description of the table DT from which it is derived, with the inclusion of a column descriptor whose column name is TRANSACTIONTIME, whose data type is a <period type> with an element precision of the transaction-time precision, and whose ordinal position is one greater than the degree of DT. The value of this additional column for each row is the original transaction-time period of the corresponding row in DT. If <value expression> is specified in the <transactiontime option> of <time option>, then the transaction-time period of the row of the result of TQEB is the value of <value expression>.

   c) Otherwise, the result of <temporal query expression body> TQEB is the result of the <query expression body> of TQEB with each of its leaf generally underlying tables with transaction-time support with no intervening <from clause> replaced with its current transaction-time value.

Language opportunity: It would be nice if <value expression> that is contained in the <transactiontime option> that is contained in <time option> also be allowed to be of a `datetime` data type, interpreted as a period containing one granule. This would allow statements of the form

```
    TRANSACTIONTIME DATE '1996-01-01' SELECT.
```

Applicability to MySQL

This change can be applied to MySQL by performing the following actions.

- In the Format, replace <transactiontime option> BNF production by the following.

```
<transactiontime option> ::=
        [ NONSEQUENCED ] TRANSACTIONTIME
            [ <left paren> <value expression> <comma>
            <value expression> <right paren> ]
```

- Replace the Syntax Rule 1 above with the following rule. The data type of <value expression> of <transactiontime option> shall be <datetime type>, with an element precision of the transaction-time precision.
- In the General Rules the period represented by the <value expression> specified in the <transactiontime option> that is contained in <time option> shall now be represented by the two <value expression> elements specified in the <transactiontime option> that is contained in <time option>, where the first element will represent the begin of the period and the second element will represent the end of the period.
- In the General Rules all references to TRANSACTIONTIME column whose data type is a <period type> with an element precision of the transaction-time precision shall now be replaced by references to columns STARTTIME and STOPTIME whose data type is a <datetime type> with an element precision of the transaction-time precision. The value of the columns would be the original begin and end of the transaction-time associated with the corresponding row.

5.2.5.2 Subclause 7.5 <query specification>

Function

Specify a table derived from the result of a <table expression>.

Format

No additional Format items.

Syntax Rules

1. (Replace SR4b) Otherwise, the <select list> ”*” is equivalent to a <value expression> sequence in which each <value expression> is a column reference that references a column of T and each column of T, other than any column named TRANSACTIONTIME, is referenced exactly once. The columns other than those named TRANSACTIONTIME are referenced in the ascending sequence of their ordinal position within T.
2. (Replace SR 5) If the <select sublist>

   <item qualifier>.*

   is specified, then let Q be the <item qualifier> of that <select sublist>. Q shall be a <table name> or <correlation name> exposed by a <table reference> immediately contained in the <from clause> of T. Let TQ be the table associated with Q. That <select sublist> is equivalent to a <value expression> sequence in which each <value expression> is a column reference CR that references a column of TQ that is not a common column of a <joined table> and does not have the name TRANSACTIONTIME. Each column of TQ that is not a referenced common column shall be referenced exactly once. The columns shall be referenced in the ascending sequence of their ordinal positions within TQ.

Note to proposal reader: This adds the TRANSACTIONTIME column to the exceptions.

Access Rules

No additional Access Rules.

General Rules

No additional General Rules.

Applicability to MySQL

This change can be applied to MySQL by adding STARTTIME and STOPTIME columns to the exceptions instead of adding the TRANSACTIONTIME column. These columns can be be accessed as discussed in Section 5.2.4.3.

5.2.6 Clause 10 Schema Definition and Manipulation

1) Insert this new Subclause, "<table definition>", to SQL/Temporal immediately following Subclause 10.1, "<default clause>".

5.2.6.1 Subclause 10.2 <table definition>

Function

Define a persistent base table, a created local temporary table, or a global temporary table.

Format

```
<table definition> ::=
        CREATE [ <table scope> ] TABLE <table name>
            { <table element list> | <subtable clause> [ <table element list> ] }
            [ <temporal definition> ]
            [ ON COMMIT <table commit action> ROWS ]
```

Note to proposal reader: This augments the production for the non-terminal <table definition> with an additional, optional clause to specify that the new table is to be a table with transaction-time support.

```
<temporal definition> ::=
        AS TRANSACTIONTIME
```

Syntax Rules

No additional Syntax Rules.

Access Rules

No additional Access Rules.

General Rules

1. (Add to GR3)

    f) Whether the table has transaction-time support or does not have transaction-time support.

Note to proposal reader: This Item is added to the table descriptor.

2. (Insert this GR) If <temporal definition> is specified, then the descriptor for the table indicates that the table has transaction-time support.

   Note to proposal reader: Otherwise, the table does not have transaction-time support.

Applicability to MySQL

This change is directly applicable to MySQL since the format followed by MySQL is similar to the Entry-level SQL-92 format. In order to allow MySQL language to support partitioned store for transaction-time databases the following changes need to be performed. In MySQL language the <table options> Subclause is contained in the <table definition> and <alter table statement> Subclauses. Hence, the following change provides the ability to create a table with a partitioned store and also alter a table to add or drop the partitioned store.

- In the Format of the MySQL language, add the following new alternative to <table options>:

  ```
  | PARTITIONED = { 0 | 1 }
  ```

  Some of the existing table options in MySQL are:

  ```
  | CHECKSUM = { 0 | 1 }
  | DELAY_KEY_WRITE = { 0 | 1 }
  ```

  Since one of our objectives is to maximize consistency with respect to MySQL language, we chose 0 and 1 as the possible values for the PARTITIONED option.

- (Insert this SR) If the value associated with PARTITIONED is 1, then <temporal definition> shall be specified.
- (Insert this GR) If the value associated with PARTITIONED is 1, then the descriptor of the table indicates that the table has a partitioned store.

5.2.6.2 Subclause 10.3 <column definition>

2) Insert this new Subclause to SQL/Temporal immediately following Subclause 10.2, "<table definition>".

Function

Define a column of a table.

Format

```
<column constraint definition> ::=
        [ <constraint name definition> ] <temporal column constraint>


<temporal column constraint> ::=
        [ <time option> ] <column constraint> [ <constraint attributes> ]
```

Note to proposal reader: This adds an optional <time option> to column constraints.

Syntax Rules

1. (Insert this SR) If TRANSACTIONTIME is specified in <time option>, then T shall be a table with transaction-time support.

2. (Insert this SR) The <value expression> that is contained in the <transactiontime option> that is contained in <time option> shall be a <literal>.

3. (Insert this SR) If TRANSACTIONTIME is specified and NONSEQUENCED is not specified in the <transactiontime option> that is contained in <time option>,

   Case:

   a) If <column constraint> is <references specification>, then the table identified by <table name> simply contained in the <referenced table and columns> of <references specification> shall be a table with transaction-time support.

   b) If <column constraint> is <check constraint definition> CCD, then for each explicit or implicit <item qualifier> IQ with a scope clause of CCD or of a <query expression> that is contained in CCD without an intervening <from clause>, IQ shall be associated with a table with transaction-time support.

Access Rules

No additional Access Rules.

General Rules

No additional General Rules.

Applicability to MySQL

This change is directly applicable to MySQL since the format followed by MySQL is similar to the Entry-level SQL-92 format. MySQL only supports UNIQUE, PRIMARY KEY, REFERENCES and NOT NULL as alternatives to <column constraint> and this change is compatible with all of them.

5.2.6.3 Subclause 10.4 <table constraint definition>

3) Insert this new Subclause to SQL/Temporal immediately following Subclause 10.3, "<column definition>".

Function

Specify an integrity constraint.

Format

```
<table constraint definition> ::=
          [ <constraint name definition> ]   [ <time option> ] <temporal table constraint>


<temporal table constraint> ::=
          <table constraint> [ <constraint attributes> ]
```

Note to proposal reader: TRANSACTIONTIME is now allowed in <time option>.
For constraints and assertions, there are four cases:

1. `CHECK`

   - works on anything
   - only considers current value

2. `TRANSACTIONTIME CHECK`
    - works only on tables with transaction-time support
    - the assertion must be true for the value at every transaction time
3. `TRANSACTIONTIME` <period exp> `CHECK`
    - like `TRANSACTIONTIME CHECK`, but only considers the times in <period exp> (a simple example is `TRANSACTIONTIME PERIOD '[1995-01-01 - 1995-12-31]' CHECK`)
4. `NONSEQUENCED TRANSACTIONTIME CHECK`
    - works on anything
    - acts like tables with transaction-time support have an explicit (unnamed) timestamp column; all rows are considered at once

`NONSEQUENCED TRANSACTIONTIME` <period exp> `CHECK` is not allowed.
End of note.

Syntax Rules

1. (Insert this SR) If TRANSACTIONTIME is specified in <time option>, then T shall be a table with transaction-time support.
2. (Insert this SR) The <value expression> that is contained in the <transactiontime option> that is contained in <time option> shall be a literal.
3. (Insert this SR) If TRANSACTIONTIME is specified and NONSEQUENCED is not specified in <table constraint definition> TCD, then for each explicit or implicit <item qualifier> IQ with a scope clause of TCD or of a <query expression> that is contained in TCD without an intervening <from clause>, IQ shall be associated with a table with transaction-time support.
4. (Insert this SR) If <transactiontime option> TO that is contained in <column constraint definition> contains NONSEQUENCED, then TO shall not contain <value expression>.

Access Rules

No additional Access Rules.

General Rules

1. (Append to GR2) The table constraint descriptor includes an indication of whether the constraint has transaction-time support or does not have transaction-time support, as well as the transaction-time period, if any, of the table constraint, if the table constraint has transaction-time support.
2. (Insert this GR) Case:
    a) If TRANSACTIONTIME is specified and NONSEQUENCED is not specified in the <transactiontime option> that is contained in <time option>, then
    Case:
        i) If <value expression> V is contained in the <transactiontime option> of <time option>, then <temporal table constraint> is satisfied if the contained <table constraint> is satisfied for each time granule TG of the value of V, with each qualified simply underlying table with transaction-time support replaced with its transaction-time value at transaction time TG.
        ii) Otherwise, <temporal table constraint> is satisfied if the contained <table constraint> is satisfied for each time granule TG in the transaction-time precision, with each leaf generally underlying table with transaction-time support with no intervening <from clause> replaced with its value at transaction time TG.

b) If NONSEQUENCED TRANSACTIONTIME is specified in <time option>, then <temporal table constraint> is satisfied if the contained <table constraint> is satisfied when each leaf generally underlying table with transaction-time support with no intervening <from clause> QTT is replaced by the table without transaction-time support comprising rows with identical values for the fields of the rows of QTT. The descriptor of that table is the same as the description of the table DT from which it is derived, with the inclusion of a column descriptor whose column name is TRANSACTIONTIME, whose data type is a <period type> with an element precision of the transaction-time precision, and whose ordinal position is one greater than the degree of DT. The value of this additional column for each row is the original transaction-time period of the corresponding row in DT.

c) Otherwise, <temporal table constraint> is satisfied if the contained <table constraint> is satisfied when each of its leaf generally underlying tables with transaction-time support with no intervening <from clause> is replaced with its current transaction-time value.

Applicability to MySQL

This change can be applied to MySQL by performing the following actions.

- In the General Rules the period represented by the <value expression> specified in the <transactiontime option> that is contained in <time option> shall now be represented by the two <value expression> elements specified in the <transactiontime option> that is contained in <time option>, where the first element will represent the begin of the period and the second element will represent the end of the period.
- In the General Rules all references to TRANSACTIONTIME column whose data type is a <period type> with an element precision of the transaction-time precision shall now be replaced by a reference to the columns STARTTIME and STOPTIME whose data type is a <datetime type> with an element precision of the transaction-time precision. The value of the columns would be the original begin and end of the transaction-time associated with the corresponding row.

MySQL supports UNIQUE, PRIMARY KEY, REFERENCES and CHECK as alternatives to <table constraint> and the above changes are compatible with all of them.

5.2.6.4 Subclause 10.5 <alter table statement>

4) Insert this new Subclause to SQL/Temporal immediately following Subclause 10.4, “<table constraint definition>”.

Function

Change the definition of a table.

Format

```
<alter table action> ::=
              !! All alternatives from ISO/EIC 9075
            | <add transaction definition>
            | <drop transaction definition>
```

Syntax Rules

1. (Insert this SR) If <add column definition>, <alter column definition>, <drop column definition>, <add supertable clause>, <drop supertable clause>, <add table constraint definition>, or <drop table constraint definition> is specified, then T shall not be a table with transaction-time support.

Language opportunity: Schema modifications of tables with transaction-time support requires versioning of the schema base tables, which will be addressed in a future change proposal.

Access Rules

No additional Access Rules.

General Rules

No additional General Rules.

Applicability to MySQL

This change is directly applicable to MySQL since the format followed by MySQL is similar to the Entry-level SQL-92 format.

5.2.6.5 Subclause 10.6 <add transaction definition>

5) Insert this new Subclause to SQL/Temporal immediately following Subclause 10.5, "<alter table statement>".

Function

Add transaction-time support to a table.

Format

```
<add transaction definition> ::=
        ADD TRANSACTIONTIME
```

Syntax Rules

1. (Insert this SR) Let T be the table identified by the <table name> that is immediately contained in the <alter table statement> that immediately contains <add transaction definition>.
2. (Insert this SR) T shall not have transaction-time support.

Access Rules

No additional Access Rules.

General Rules

1. (Insert this GR) Transaction-time support is added to each row R of T, by associating with R a transaction time P such that BEGIN(P) is CURRENT_TIMESTAMP and END(P) is the end of time in the transaction-time precision. The descriptor of T is altered to indicate that T has transaction-time support.
2. (Insert this GR) If T is a supertable, then an <add transaction definition>, without further Access Rule checking, is effectively performed for each of its subtables, thereby adding transaction-time support in these subtables.

Applicability to MySQL

This change is directly applicable to MySQL as it does not conflict with the existing features in MySQL. The ability to add the partitioned store to a table has been achieved by the changes proposed to Subclause <table options> in Section 5.2.6.1.

5.2.6.6 Subclause 10.7 <drop transaction definition>

1) Insert this new Subclause to SQL/Temporal immediately following Subclause 10.6, "<add transaction definition>".

Function

Drop transaction-time support from a table.

Format

```
<drop valid definition> ::=
        DROP TRANSACTIONTIME
```

Syntax Rules

1. (Insert this SR) Let T be the table identified by the <table name> that is immediately contained in the <alter table statement> that immediately contains <drop transaction definition>.
2. (Insert this SR) T shall be a table with transaction-time support.

Access Rules

No additional Access Rules.

General Rules

1. (Insert this GR) Case:

   a) If T has valid-time support, then transaction-time support is removed from T by replacing T with the result of
      VALIDTIME SELECT * FROM T

   b) Otherwise, transaction-time support is removed from T by replacing T with the result of
      SELECT * FROM T

   The descriptor of T is altered to indicate that T does not have transaction-time support.

   Note to proposal reader: That is, only the current state is retained. Previously stored transaction-time values are no longer accessible.

2. (Insert this GR) If T is a supertable, then let ST be the <table name> of any subtable of T. The following <alter table statement> is effectively executed without further Access Rule checking:
   ALTER TABLE ST DROP TRANSACTIONTIME

Applicability to MySQL

This change is directly applicable to MySQL as it does not conflict with the existing features in MySQL. The ability to drop the partitioned store from a table has been achieved by the changes proposed to Subclause <table options> in Section 5.2.6.1.

5.2.6.7 Subclause 10.8 <assertion definition>

8) Insert this new Subclause to SQL/Temporal immediately following Subclause 10.7, "<drop transaction definition>".

Function

Specify an integrity constraint by means of an assertion and specify when the assertion is to be checked.

Format

<triggered assertion> ::=
    [<time option>]
        CHECK <left paren> <search condition> <right paren>

Note to proposal reader: This adds an optional <time option>.

Syntax Rules

1. (Insert this SR) If TRANSACTIONTIME is specified and NONSEQUENCED is not specified in the <transactiontime option> that is contained in <time option>, then for each explicit or implicit <item qualifier> IQ with a scope clause of the <search condition> SC that is simply contained in <triggered assertion> or of a <query expression> that is contained in SC without an intervening <from clause>, IQ shall be associated with a table with transaction-time support.
2. (Insert this SR) The <value expression> contained in the <transactiontime option> contained in <time option> shall be a <literal>.

Access Rules

No additional Access Rules.

General Rules

1. (Append to GR4) The assertion descriptor includes an indication of whether the assertion has transaction-time support or does not have transaction-time support, as well as the transaction-time period, if any, of the assertion, if the assertion has transaction-time support.

2. (Insert this GR) Case:
   a) If TRANSACTIONTIME is specified and NONSEQUENCED is not specified in the <transactiontime option> that is contained in <time option>, then
      Case:
      i) If <value expression> V is contained in the <transactiontime option> that is contained in <time option>, then <triggered assertion> is satisfied if the contained <search condition> is satisfied for each time granule TG of the value of V, with each leaf generally underlying table with transaction-time support with no intervening <from clause> replaced with its state at transaction time TG.
      ii) Otherwise, <triggered assertion> is satisfied if the contained <search condition> is satisfied for each time granule TG of the transaction-time precision, with each leaf generally underlying table with transaction-time support with no intervening <from clause> replaced with its state at transaction time TG.
   b) If NONSEQUENCED TRANSACTION is specified in <time option>, then <triggered assertion> is satisfied if the contained <search condition> is satisfied when each leaf generally underlying table with transaction-time support with no intervening <from clause> QTT is replaced by the table without transaction-time support comprising rows with identical values for the fields of the rows of QTT. The descriptor of that table is the same as the description of the table DT from which it is derived, with the inclusion of a column descriptor whose column name is TRANSACTIONTIME, whose data type is a <period type> with an element precision of the transaction-time precision, and whose ordinal position is one greater than the degree of DT. The value of this additional column for each row is the original transaction-time period of the corresponding row in DT.
   c) Otherwise, <triggered assertion> is satisfied if the contained <search condition> is satisfied when each of its leaf generally underlying tables with transaction-time support with no intervening <from clause> is replaced with its current transaction-time value.

Applicability to MySQL

This change shall not be applied to MySQL as it does not have the ability to define assertions.

7) Insert this new Subclause, "<check constraint definition>", to SQL/Temporal immediately following Subclause 10.9, "<assertion definition>".

5.2.6.8 Subclause 10.10 <check constraint definition>

Function

Specify a condition for the SQL-data.

Format

No additional Format Items.

Syntax Rules

1. (Insert this SR) Any <query expression> simply contained in <search condition> without an intervening <from clause> shall not simply contain a <time option>.

Access Rules

No additional Access Rules.

General Rules

No additional General Rules.

Applicability to MySQL

The changes proposed in the Subclause 10.10 "<check constraint definition>" are directly applicable to MySQL.

5.2.7 Clause 12 Data Manipulation

1) Insert the following new Subclause, "<select statement: single row>", to SQL/Temporal at the beginning of Clause 12, "Date manipulation".

5.2.7.1 Subclause 12.2 <select statement: single row>

Function

Retrieve values from a specified row of a table.

Format

```
<select statement: single row> ::=
            [ <time option> ]
            SELECT [ <set quantifier> ] <select list>
            INTO <select target list>
            <table expression>
```

Note to proposal reader: This adds an optional <time option>.

Syntax Rules

1. (Insert this SR) If TRANSACTIONTIME is specified and NONSEQUENCED is not specified in the <transactiontime option> that is contained in <time option>, then for each explicit or implicit <item qualifier> IQ with a scope clause of the <table expression> TE or of a <query expression> that is contained in TE without an intervening <from clause>, IQ shall be associated with a table with transaction-time support.
2. (Insert this SR) If TRANSACTIONTIME is specified in a <time option> of a <query expression> Q that is contained in the <table expression> of <select statement: single row>, then Q shall be simply contained in a <from clause>.
3. (Insert this SR) Case:
    a) If TRANSACTIONTIME is specified and NONSEQUENCED is not specified in the <transactiontime option> that is contained in <time option>, then T shall be a table with transaction-time support.
    b) If NONSEQUENCED TRANSACTIONTIME is specified in <time option>, then
       Case:
        i) If <value expression> is specified in the <transactiontime option> of <time option>, then T shall be a table with transaction-time support.
        ii) Otherwise, T shall be a table without transaction-time support.
    c) Otherwise, T shall be a table without transaction-time support.

   Note to proposal reader: Subclause 6.1 "<item reference>" restricts the scope of column names in the <value expression> that is contained in the <transactiontime option> that is contained in the <time option>.

Access Rules

No additional Access Rules.

General Rules

1. (Insert this GR) Case:
    a) If TRANSACTIONTIME is specified and NONSEQUENCED is not specified in the <transactiontime option> that is contained in <time option>, then the result of <table expression> TE during each transaction time granule TG of the transaction-time precision is the result of TE, in accordance with General Rule 6 of this Subclause, with each qualified simply underlying table with transaction-time support replaced with its transaction-time value at transaction time TG. If <value expression> VE is specified in the <transactiontime option> that is contained in <time option>, then for each row R resulting from the initial result, that is, before the following replacements, of TE,
       Case:
        i) If the value of VE and the transaction-time period VP of R overlap, then the resulting transaction-time period of R is the result of
           (VE P_INTERSECT VP).
        ii) Otherwise, R is not included in the final result of TE.
    b) If NONSEQUENCED TRANSACTIONTIME is specified in <time option>, then the the result of <table expression> TE is the result of TE, in accordance with General Rule 6 of this Subclause, with each leaf generally underlying table with transaction-time support with no intervening <from clause> QTT replaced by the table without transaction-time support comprising rows with identical

values for the fields of the rows of QTT. The descriptor of that table is the same as the description of the table DT from which it is derived, with the inclusion of a column descriptor whose column name is TRANSACTIONTIME, whose data type is a <period type> with an element precision of the transaction-time precision, and whose ordinal position is one greater than the degree of DT. The value of this additional column for each row is the original transaction-time period of the corresponding row in DT. If <value expression> is specified in the <transactiontime option> of <time option>, then the transaction-time period of the row of the result has the value of <value expression>.

c) Otherwise, the result of <table expression> TE is the result of TE, in accordance with General Rule 6 of this Subclause, with each of its leaf generally underlying tables with transaction-time support with no intervening <from clause> replaced with its current transaction-time value.

Applicability to MySQL

This change can be applied to MySQL by performing the following actions.

- In the General Rules the period represented by the <value expression> specified in the <transactiontime option> that is contained in <time option> shall now be represented by the two <value expression> elements specified in the <transactiontime option> that is contained in <time option>, where the first element will represent the begin of the period and the second element will represent the end of the period.
- In the General Rules all references to TRANSACTIONTIME column whose data type is a <period type> with an element precision of the transaction-time precision shall now be replaced by a reference to the columns STARTTIME and STOPTIME whose data type is a <datetime type> with an element precision of the transaction-time precision. The value of the columns would be the original begin and end of the transaction-time associated with the corresponding row.

5.2.7.2 Subclause 12.3 <delete statement: searched>

4) Insert this new Subclause to SQL/Temporal immediately following Subclause 12.2, “<select statement: single row>”.

Function

Delete rows of a table.

Format

```
<delete statement: searched> ::=
        [ <time option> ]
        DELETE FROM <table reference>
            [ WHERE <search condition> ]
```

Note to proposal reader: This augments the production for <delete statement: searched> with an additional, optional clause. TRANSACTIONTIME is now allowed in <time option>.

Syntax Rules

1. (Insert this SR) Let T be the subject table of the <delete statement: searched>.
2. (Insert this SR) If TRANSACTIONTIME is specified in <time option>, then T shall be a table with transaction-time support.
3. (Insert this SR) If TRANSACTIONTIME is specified in a <time option> of a <query expression> Q that is contained in the <search condition> of <delete statement: searched>, then Q shall be simply contained in a <from clause>.

4. (Insert this SR) A <value expression> shall not be contained in the <transactiontime option> of <time option>.
5. (Insert this SR) The scope of the <table reference> is the entire <delete statement: searched>.

Access Rules

No additional Access Rules.

General Rules

1. (Insert this GR) Case:
    a) If NONSEQUENCED TRANSACTIONTIME is specified in <time option>, then the <search condition> SC is satisfied if SC is satisfied, in accordance with General Rule 13 of this Subclause, when each qualified simply underlying table with transaction-time support QTT is replaced by the table without transaction-time support comprising rows with identical values for the fields of the rows of QTT. The descriptor of that table is the same as the description of the table DT from which it is derived, with the inclusion of a column descriptor whose column name is TRANSACTIONTIME, whose data type is a <period type> with an element precision of the transaction-time precision, and whose ordinal position is one greater than the degree of DT. The value of this additional column for each row is the original transaction-time period of the corresponding row in DT. If the <search condition> is satisfied, then the row is marked for deletion.
    b) Otherwise, the <search condition> SC is satisfied if SC is satisfied, in accordance with General Rule 13 of this Subclause, when each of its leaf generally underlying tables with transaction-time support with no intervening <from clause> is replaced with its current transaction-time value. If the <search condition> is satisfied for the relevant row, then the row is marked for deletion.
2. (Insert this GR) If T is a table with transaction-time support, then to logically delete a row, the ending bound of the transaction time of the row is set to CURRENT_TIMESTAMP in the transaction-time precision.

Applicability to MySQL

The changes to the syntax are directly applicable to MySQL since the format followed by MySQL is similar to the Entry-level SQL-92 format. The General Rules can be directly applied to MySQL after performing the following action.

- In the General Rules all references to TRANSACTIONTIME column whose data type is a <period type> with an element precision of the transaction-time precision shall now be replaced by a reference to the columns STARTTIME and STOPTIME whose data type is a <datetime type> with an element precision of the transaction-time precision. The value of the columns would be the original begin and end of the transaction-time associated with the corresponding row.

5.2.7.3 Subclause 12.4 <delete statement: positioned>

1) Add the following Syntax Rule:

1. (Insert this SR) TRANSACTIONTIME shall not be specified in <time option>.

2) Add the following General Rule:

1. (Insert this GR) If T is a table with transaction-time support, then to logically delete a row, the ending bound of the transaction time of the row is set to CURRENT_TIMESTAMP in the transaction-time precision.

Applicability to MySQL

This change shall not be applied to MySQL as it does not have positioned delete.

5.2.7.4 Subclause 12.5 <insert statement>

5) Insert this new Subclause to SQL/Temporal immediately following Subclause 12.3, "<delete statement: searched>".

Function

Create new rows in a table.

Format

No additional Format items.

Syntax Rules

1. (Insert this SR) The result of <insert columns and source> shall be a table without transaction-time support.

Access Rules

No additional Access Rules.

General Rules

1. (Insert this GR) If T is a table with transaction-time support, then each row of the result of <insert columns and source> shall be associated with a transaction time P such that BEGIN(P)is CURRENT_TIMESTAMP and END(P) is the end of time in the transaction-time precision.

5.2.7.5 Subclause 12.6 <update statement: searched>

7) Insert this new Subclause to SQL/Temporal immediately following Subclause 12.4, "<insert statement>".

Function

Update rows of a table.

Format

```
<update statement: searched> ::=
          [ <time option> ]
          UPDATE <table reference>
               SET <set clause list>
               [ WHERE <search condition> ]
```

Note to proposal reader: This adds an optional <time option>.

Syntax Rules

1. (Insert this SR) If TRANSACTIONTIME is specified in <time option>, then T shall be a table with transaction-time support.
2. (Insert this SR) If TRANSACTIONTIME is specified in a <time option> of a <query expression> Q that is contained in the <search condition> of <update statement: searched>, then Q shall be simply contained in a <from clause>.
3. (Insert this SR) A <value expression> shall not be contained in the <transactiontime option> of <time option>.

Access Rules

No additional Access Rules.

General Rules

1. (Insert this GR) Case:

   a) If NONSEQUENCED TRANSACTIONTIME is specified in <time option>, then the <search condition> SC is satisfied if SC is satisfied, in accordance with General Rule 23 of this Subclause, when each leaf generally underlying table with transaction-time support with no intervening <from clause> is replaced with a table with no transaction-time support with rows with identical values for the columns. The descriptor of that table is the same as the description of the table DT from which it is derived, with the inclusion of a column descriptor whose column name is TRANSACTIONTIME, whose data type is a <period type> with an element precision of the transaction-time precision, and whose ordinal position is one greater than the degree of DT. The value of this additional column for each row is the original transaction-time period of the corresponding row in DT.

   b) Otherwise, the <search condition> SC is satisfied if SC is satisfied, in accordance with General Rule 23 of this Subclause, when each of its leaf generally underlying tables with transaction-time support with no intervening <from clause> is replaced with its current transaction-time state.

2. (Insert this GR) If T is a table with transaction-time support, the ending bound of the transaction time of the current row is set to CURRENT_TIMESTAMP in the transaction-time precision. Let NR be a row with column values identical to the current row, with an associated transaction time P such that BEGIN(P) is CURRENT_TIMESTAMP and END(P) is the end of time in the transaction-time precision. Perform the update on NR, then insert NR into T.

Applicability to MySQL

The changes to the syntax are directly applicable to MySQL since the format followed by MySQL is similar to the Entry-level SQL-92 format. The General Rules can be directly applied to MySQL after performing the following action.

- In the General Rules all references to TRANSACTIONTIME column whose data type is a <period type> with an element precision of the transaction-time precision shall now be replaced by a reference to the columns STARTTIME and STOPTIME whose data type is a <datetime type> with an element precision of the transaction-time precision. The value of the columns would be the original begin and end of the transaction-time associated with the corresponding row.

5.2.7.6 Subclause 12.7 <update statement: positioned>

1) Add the following Syntax Rule:

1. (Insert this SR) TRANSACTIONTIME shall not be specified in <time option>.

2) Add the following General Rule:

1. (Insert this GR) If T is a table with transaction-time support, the ending bound of the transaction time of the current row is set to CURRENT_TIMESTAMP in the transaction-time precision. Let NR be a row with column values identical to the current row, with an associated transaction time P such that BEGIN(P) is CURRENT_TIMESTAMP and END(P) is the end of time in the transaction-time precision. Perform the update on NR, then insert NR into T.

Applicability to MySQL

This change shall not be applied to MySQL as it does not have positioned update.

5.2.8 Clause 12 Information Schema and Definition Schema

5.2.8.1 Subclause 12.1 Information Schema

1) Insert the following new Table, “TABLES view”, to SQL/Temporal immediately preceding Subclause 12.2, “Definition Schema”.

Subclause 12.1.1 TABLES view

Function

Identify the tables defined in this catalog that are accessible to a given user.

Definition

```
CREATE VIEW TABLES
AS SELECT
       TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME, TABLE_TYPE,
       TRANSACTIONTIME_SUPPORT
FROM DEFINITION_SCHEMA.TABLES
    WHERE ( TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME )
       IN (
   SELECT TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME
    FROM DEFINITION_SCHEMA.TABLE_PRIVILEGES
      WHERE GRANTEE IN ( ’PUBLIC’, CURRENT_USER )
   UNION
   SELECT TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME
    FROM DEFINITION_SCHEMA.COLUMN_PRIVILEGES
      WHERE GRANTEE IN ( ’PUBLIC’, CURRENT_USER ) )
       AND TABLE_CATALOG
= ( SELECT CATALOG_NAME FROM INFORMATION_SCHEMA_CATALOG_NAME )
```

Note to proposal reader: This adds one column: TRANSACTIONTIME_SUPPORT.

Leveling Rules

No additional Leveling Rules.

Subclause 12.1.2 VIEWS view

1) Insert this new Table to SQL/Temporal immediately following Subclause 12.1.1, “TABLES view”.

Function

Identify the viewed tables defined in this catalog that are accessible to a given user.

Definition

```
CREATE VIEW VIEWS
      AS SELECT
TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME,
CASE WHEN ( TABLE_CATALOG, TABLE_SCHEMA, CURRENT_USER )
    IN ( SELECT CATALOG_NAME, SCHEMA_NAME, SCHEMA_OWNER
FROM DEFINITION_SCHEMA.SCHEMATA )
     THEN VIEW_DEFINITION
     ELSE NULL
END AS VIEW_DEFINITION,
CHECK_OPTION, IS_UPDATABLE,
TRANSACTIONTIME_SUPPORT
      FROM DEFINITION_SCHEMA.VIEWS
  WHERE ( TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME )
    IN ( SELECT TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME
 FROM TABLES )
  AND TABLE_CATALOG
    = ( SELECT CATALOG_NAME FROM INFORMATION_SCHEMA_CATALOG_NAME )
```

Note to proposal reader: This adds one column: TRANSACTIONTIME_SUPPORT.

Leveling Rules

No additional Leveling Rules.

Subclause 12.1.3 TABLE_CONSTRAINTS view

1) Insert this new Table to SQL/Temporal immediately following Subclause 12.1.2, “VIEWS view”.

Function

Identify the table constraints defined in this catalog that are owned by a given user.

Definition

```
CREATE VIEW TABLE_CONSTRAINTS
      AS SELECT
  CONSTRAINT_CATALOG, CONSTRAINT_SCHEMA, CONSTRAINT_NAME,
  TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME,
  CONSTRAINT_TYPE, IS_DEFERRABLE, INITIALLY_DEFERRED,
  TRANSACTIONTIME_SUPPORT, TRANSACTIONTIME_PERIOD
FROM DEFINITION_SCHEMA.TABLE_CONSTRAINTS
     JOIN
     DEFINITION_SCHEMA.SCHEMATA S
     ON
      ( ( CONSTRAINT_CATALOG, CONSTRAINT_SCHEMA )
      = ( S.CATALOG_NAME, S.SCHEMA_NAME ) )
   WHERE SCHEMA_OWNER = CURRENT_USER
  AND CONSTRAINT_CATALOG
    = ( SELECT CATALOG_NAME FROM INFORMATION_SCHEMA_CATALOG_NAME )
```

Note to proposal reader: This adds two columns: TRANSACTIONTIME_SUPPORT and TRANSACTION-TIME_PERIOD.

Leveling Rules

No additional Leveling Rules.

Subclause 12.1.4 ASSERTIONS view

1) Insert this new Table to SQL/Temporal immediately following Subclause 12.1.3, “TABLE_CONSTRAINTS view”.

Function

Identify the assertions defined in this catalog that are owned by a given user.

Definition

```
CREATE VIEW ASSERTIONS
      AS SELECT
  CONSTRAINT_CATALOG, CONSTRAINT_SCHEMA, CONSTRAINT_NAME,
  IS_DEFERRABLE, INITIALLY_DEFERRED,
  TRANSACTIONTIME_SUPPORT, TRANSACTIONTIME_PERIOD
FROM DEFINITION_SCHEMA.ASSERTIONS
     JOIN
     DEFINITION_SCHEMA.SCHEMATA S
     ON
      ( ( CONSTRAINT_CATALOG, CONSTRAINT_SCHEMA )
      = ( S.CATALOG_NAME, S.SCHEMA_NAME ) )
   WHERE SCHEMA_OWNER = CURRENT_USER
  AND CONSTRAINT_CATALOG
    = ( SELECT CATALOG_NAME FROM INFORMATION_SCHEMA_CATALOG_NAME )
```

Note to proposal reader: This adds two columns: TRANSACTIONTIME_SUPPORT and TRANSACTION-TIME_PERIOD.

Leveling Rules

No additional Leveling Rules.

Applicability to MySQL

The changes suggested in Clause 12.1 "Information Schema" shall not be applied MySQL as it does not support views.

5.2.8.2 Subclause 12.2 Definition Schema

1) Insert the following new Table, "TABLES base table", to SQL/Temporal immediately following Subclause 12.2.1, "DATA_TYPE_DESCRIPTOR base table".

Subclause 12.2.2 TABLES base table

Function

The TABLES table contains one row for each table including views. It effectively contains a representation of the table descriptors.

Definition

```
CREATE TABLE TABLES
 (
 TABLE_CATALOG         INFORMATION_SCHEMA.SQL_IDENTIFIER,
 TABLE_SCHEMA          INFORMATION_SCHEMA.SQL_IDENTIFIER,
 TABLE_NAME            INFORMATION_SCHEMA.SQL_IDENTIFIER,
 TABLE_TYPE            INFORMATION_SCHEMA.CHARACTER_DATA
       CONSTRAINT TABLE_TYPE_NOT_NULL NOT NULL,
       CONSTRAINT TABLE_TYPE_CHECK CHECK ( TABLE_TYPE IN
       ( 'BASE TABLE', 'VIEW', 'GLOBAL TEMPORARY',
 'LOCAL TEMPORARY' ) ),
       CONSTRAINT CHECK_TABLE_IN_COLUMNS
CHECK ( ( TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME ) IN
      ( SELECT TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME
FROM COLUMNS ) ),
 TRANSACTIONTIME_SUPPORT INFORMATION_SCHEMA.CHARACTER_DATA
       CONSTRAINT TRANSACTIONTIME_SUPPORT_CHECK
 CHECK (TRANSACTIONTIME_SUPPORT IN ('STATE','NONE')),

 CONSTRAINT TABLES_PRIMARY_KEY
       PRIMARY KEY ( TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME ),

 CONSTRAINT TABLES_FOREIGN_KEY_SCHEMATA
       FOREIGN KEY ( TABLE_CATALOG, TABLE_SCHEMA ) REFERENCES SCHEMATA,

 CONSTRAINT TABLES_CHECK_NOT_VIEW CHECK ( NOT EXISTS
       ( SELECT TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME
   FROM TABLES
   WHERE TABLE_TYPE = 'VIEW'
 EXCEPT
 SELECT TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME
   FROM VIEWS ) )
 )
```

Note to proposal reader: This adds one column: TRANSACTIONTIME_SUPPORT.

Description

1. The values of TRANSACTIONTIME_SUPPORT have the following meanings:

   STATE The table being described has transaction-time support.

   NONE The table being described does not have transaction-time support.

Subclause 12.2.3 VIEWS base table

1) Insert this new Table to SQL/Temporal immediately following Subclause 12.2.2, “TABLES base table”.

Function

The VIEWS table contains one row for each row in the TABLES table with a TABLE_TYPE of 'VIEW'. Each row describes the query expression that defines a view. The table effectively contains a representation of the view descriptors.

Definition

```
CREATE TABLE VIEWS
 (
 TABLE_CATALOG      INFORMATION_SCHEMA.SQL_IDENTIFIER,
 TABLE_SCHEMA       INFORMATION_SCHEMA.SQL_IDENTIFIER,
 TABLE_NAME         INFORMATION_SCHEMA.SQL_IDENTIFIER,
 VIEW_DEFINITION    INFORMATION_SCHEMA.CHARACTER_DATA,
 CHECK_OPTION       INFORMATION_SCHEMA.CHARACTER_DATA
```

```
      CONSTRAINT CHECK_OPTION_NOT_NULL NOT NULL
      CONSTRAINT CHECK_OPTION_CHECK
CHECK ( CHECK_OPTION IN ( 'CASCADED', 'LOCAL', 'NONE' ) ),
IS_UPDATABLE       INFORMATION_SCHEMA.CHARACTER_DATA
      CONSTRAINT IS_UPDATABLE_NOT_NULL NOT NULL
      CONSTRAINT IS_UPDATABLE_CHECK CHECK ( IS_UPDATABLE IN ( 'YES', 'NO' ) ),
TRANSACTIONTIME_SUPPORT INFORMATION_SCHEMA. CHARACTER_DATA
      CONSTRAINT TRANSACTIONTIME_SUPPORT_CHECK
CHECK (TRANSACTIONTIME_SUPPORT IN ('STATE','NONE')),

CONSTRAINT VIEWS_PRIMARY_KEY
      PRIMARY KEY ( TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME ),

CONSTRAINT VIEWS_IN_TABLES_CHECK
      CHECK ( ( TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME ) IN
      ( SELECT TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME
  FROM TABLES
  WHERE TABLE_TYPE = 'VIEW' ) ),

CONSTRAINT VIEWS_IS_UPDATABLE_CHECK_OPTION_CHECK
      CHECK ( ( IS_UPDATABLE, CHECK_OPTION ) NOT IN
( VALUES ( 'NO', 'CASCADED' ), ( 'NO', 'LOCAL' ) ) )
)
```

Note to proposal reader: This adds one column: TRANSACTIONTIME_SUPPORT.

Description

1. The values of TRANSACTIONTIME_SUPPORT have the following meanings:

   STATE The table being described has transaction-time support.

   NONE The table being described does not have transaction-time support.

Subclause 12.2.4 TABLE_CONSTRAINTS base table

1) Insert this new Table to SQL/Temporal immediately following Subclause 12.2.3, “VIEWS base table”.

Function

The TABLE_CONSTRAINTS table has one row for each table constraint associated with a table. It effectively contains a representation of the table constraint descriptors.

Definition

```
CREATE TABLE TABLE_CONSTRAINTS
 (
 CONSTRAINT_CATALOG      INFORMATION_SCHEMA.SQL_IDENTIFIER,
 CONSTRAINT_SCHEMA       INFORMATION_SCHEMA.SQL_IDENTIFIER,
 CONSTRAINT_NAME         INFORMATION_SCHEMA.SQL_IDENTIFIER,
 CONSTRAINT_TYPE         INFORMATION_SCHEMA.CHARACTER_DATA
 CONSTRAINT CONSTRAINT_TYPE_NOT_NULL NOT NULL
       CONSTRAINT CONSTRAINT_TYPE_CHECK
       CHECK ( CONSTRAINT_TYPE IN
       ( 'UNIQUE',
  'PRIMARY KEY',
  'FOREIGN KEY',
  'CHECK' ) ),

 TABLE_CATALOG             INFORMATION_SCHEMA.SQL_IDENTIFIER
       CONSTRAINT TABLE_CONSTRAINTS_TABLE_CATALOG_NOT_NULL NOT NULL,
 TABLE_SCHEMA              INFORMATION_SCHEMA.SQL_IDENTIFIER
       CONSTRAINT TABLE_CONSTRAINTS_TABLE_SCHEMA_NOT_NULL NOT NULL,
 TABLE_NAME                INFORMATION_SCHEMA.SQL_IDENTIFIER
       CONSTRAINT TABLE_CONSTRAINTS_TABLE_NAME_NOT_NULL NOT NULL,
 IS_DEFERRABLE             INFORMATION_SCHEMA.CHARACTER_DATA
       CONSTRAINT TABLE_CONSTRAINTS_IS_DEFERRABLE_NOT_NULL NOT NULL,
 INITIALLY_DEFERRED        INFORMATION_SCHEMA.CHARACTER_DATA
       CONSTRAINT TABLE_CONSTRAINTS_INITIALLY_DEFERRED_NOT_NULL
 TRANSACTIONTIME_SUPPORT  INFORMATION_SCHEMA.CHARACTER_DATA
       CONSTRAINT TRANSACTIONTIME_SUPPORT_CHECK
 CHECK (TRANSACTIONTIME_SUPPORT IN ('SEQUENCED','NONSEQUENCED','NONE')),
 TRANSACTIONTIME_PERIOD   INFORMATION_SCHEMA.CARDINAL_NUMBER,

 CONSTRAINT TABLE_CONSTRAINTS_PRIMARY_KEY
       PRIMARY KEY ( CONSTRAINT_CATALOG, CONSTRAINT_SCHEMA, CONSTRAINT_NAME ),

 CONSTRAINT TABLE_CONSTRAINTS_DEFERRED_CHECK
       CHECK ( ( IS_DEFERRABLE, INITIALLY_DEFERRED ) IN
      ( VALUES ( 'NO',  'NO' ),
      ( 'YES', 'NO' ),
      ( 'YES', 'YES' ) ) ),

 CONSTRAINT TABLE_CONSTRAINTS_CHECK_VIEWS
       CHECK ( TABLE_CATALOG
      <> ANY ( SELECT CATALOG_NAME FROM SCHEMATA )
  OR
    ( ( TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME ) IN
      ( SELECT TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME
FROM TABLES
WHERE TABLE_TYPE <> 'VIEW' ) ) ),

 CONSTRAINT TABLE_CONSTRAINTS_UNIQUE_CHECK
       CHECK ( 1 =
 ( SELECT COUNT (*)
   FROM ( SELECT CONSTRAINT_CATALOG, CONSTRAINT_SCHEMA, CONSTRAINT_NAME
  FROM TABLE_CONSTRAINTS
  WHERE CONSTRAINT_TYPE IN ( 'UNIQUE', 'PRIMARY KEY' )
   UNION ALL
   SELECT CONSTRAINT_CATALOG, CONSTRAINT_SCHEMA, CONSTRAINT_NAME
   FROM REFERENTIAL_CONSTRAINTS
   UNION ALL
   SELECT CONSTRAINT_CATALOG, CONSTRAINT_SCHEMA, CONSTRAINT_NAME
   FROM CHECK_CONSTRAINTS ) AS X
   WHERE ( CONSTRAINT_CATALOG, CONSTRAINT_SCHEMA, CONSTRAINT_NAME )
       = ( X.CONSTRAINT_CATALOG, X.CONSTRAINT_SCHEMA, X.CONSTRAINT_NAME ) ) ),

 CONSTRAINT UNIQUE_TABLE_PRIMARY_KEY_CHECK
       CHECK ( UNIQUE ( SELECT TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME
FROM TABLE_CONSTRAINTS
WHERE CONSTRAINT_TYPE = 'PRIMARY KEY' ) )
 )
```

Note to proposal reader: This adds two columns: TRANSACTIONTIME_SUPPORT and TRANSACTIONTIME_PERIOD.

Description

1. The values of TRANSACTIONTIME_SUPPORT have the following meanings:

   SEQUENCED The table constraint being described was specified with TRANSACTIONTIME and without NONSEQUENCED.

   NONSEQUENCED The table constraint being described was specified with NONSEQUENCED TRANSACTIONTIME.

   NONE TRANSACTIONTIME was not specified in the table constraint being described.

2. The value of TRANSACTIONTIME_PERIOD is the value of the <value expression> contained in the <transactiontime option> associated with the table constraint being described.

Subclause 12.2.5 ASSERTIONS base table

1) Insert this new Table to SQL/Temporal immediately following Subclause 12.2.4, "TABLE_CONSTRAINTS base table".

Function

The ASSERTIONS table has one row for each assertion. It effectively contains a representation of the assertion descriptors.

Definition

```
CREATE TABLE ASSERTIONS
 (
 CONSTRAINT_CATALOG      INFORMATION_SCHEMA.SQL_IDENTIFIER,
 CONSTRAINT_SCHEMA       INFORMATION_SCHEMA.SQL_IDENTIFIER,
 CONSTRAINT_NAME         INFORMATION_SCHEMA.SQL_IDENTIFIER,
 IS_DEFERRABLE           INFORMATION_SCHEMA.CHARACTER_DATA
       CONSTRAINT ASSERTIONS_IS_DEFERRABLE_NOT_NULL NOT NULL,
 INITIALLY_DEFERRED      INFORMATION_SCHEMA.CHARACTER_DATA
       CONSTRAINT ASSERTIONS_INITIALLY_DEFERRED_NOT_NULL NOT NULL,
 CHECK_TIME              INFORMATION_SCHEMA.CHARACTER_DATA
       CONSTRAINT ASSERTIONS_CHECK_TIME_CHECK
  CHECK ( CHECK_TIME IN ('IMMEDIATE', 'DEFERRED' ) ),
 TRANSACTIONTIME_SUPPORT INFORMATION_SCHEMA.CHARACTER_DATA
       CONSTRAINT TRANSACTIONTIME_SUPPORT_CHECK
   CHECK (TRANSACTIONTIME_SUPPORT IN ('SEQUENCED','NONSEQUENCED','NONE')),
 TRANSACTIONTIME_PERIOD  INFORMATION_SCHEMA.CARDINAL_NUMBER,

 CONSTRAINT ASSERTIONS_PRIMARY_KEY
       PRIMARY KEY ( CONSTRAINT_CATALOG, CONSTRAINT_SCHEMA, CONSTRAINT_NAME ),

 CONSTRAINT ASSERTIONS_FOREIGN_KEY_CHECK_CONSTRAINTS
       FOREIGN KEY (CONSTRAINT_CATALOG, CONSTRAINT_SCHEMA, CONSTRAINT_NAME )
 REFERENCES CHECK_CONSTRAINTS,

 CONSTRAINT ASSERTIONS_FOREIGN_KEY_SCHEMATA
       FOREIGN KEY ( CONSTRAINT_CATALOG, CONSTRAINT_SCHEMA )
 REFERENCES SCHEMATA,

 CONSTRAINT ASSERTIONS_DEFERRED_CHECK
       CHECK ( ( IS_DEFERRABLE, INITIALLY_DEFERRED ) IN
  VALUES ( ( 'NO',  'NO' ),
   ( 'YES', 'NO' ),
   ( 'YES', 'YES' ) ) )
  )
```

Note to proposal reader: This adds two columns: TRANSACTIONTIME_SUPPORT and TRANSACTION-TIME_PERIOD.

Description

1. The values of TRANSACTIONTIME_SUPPORT have the following meanings:

   SEQUENCED The assertion being described was specified with TRANSACTIONTIME and without NON-SEQUENCED.

   NONSEQUENCED The assertion being described was specified with NONSEQUENCED TRANSAC-TIONTIME.

   NONE TRANSACTIONTIME was not specified in the assertion being described.

2. The value of TRANSACTIONTIME_PERIOD is the value of the <value expression> contained in the <transactiontime option> associated with the assertion being described.

Applicability to MySQL

MySQL maintains schema information associated to tables in files named as *table_name*`.frm`. The changes suggested in Clause 12.2 "Definition Schema" can be applied to MySQL by adding transaction-time and partitioned store related information to the metadata maintained in the *table_name*`.frm` file.
The schema information stored in MySQL can be seen by using commands `SHOW TABLES`, which lists the tables in the database, `SHOW COLUMNS FROM` *table_name* and `DESC` *table_name*, which list the columns in the table along with their attributes. The result of these commands shall not be altered in order to maintain temporal upward compatibility. We propose the command `SHOW TRANSACTIONTIME TABLES`, which shall list the tables in the database along with transaction-time and partitioned store related information associated to the tables. The result set of this command shall contain columns `Table_name`, `Transactiontime` and `Partitioned` of character data type. The columns `Transactiontime` and `Partitioned` in the result set shall have a value `YES` if the associated table has transaction-time support and partitioned store support respectively. Otherwise the columns `Transactiontime` and `Partitioned` shall contain an empty string. The new commands `SHOW COLUMNS FROM TRANSACTIONTIME` *table_name* and `DESC TRANSACTIONTIME` *table_name* shall return the same result set. Both commands shall show the transaction-time related information associated to the columns of the table, along with the other attributes of the columns. The result set shall contain the extra columns `Null_transactiontime` and `Key_transactiontime` of character data type. The column `Null_transactiontime` in the result set shall have a value `SEQUENCED` or `NONSEQUENCED` or an empty string based on whether the `NOT NULL` constraint associated to the corresponding column of the table is sequenced or nonsequenced or neither. The column `Key_transactiontime` in the result set shall have a value `SEQUENCED` or `NONSEQUENCED` or an empty string based on whether the `PRIMARY KEY` or `UNIQUE` constraint associated to the corresponding column of the table is sequenced or nonsequenced or neither. The result set shall also contain the columns `Null_transactiontime_start`, `Null_transactiontime_stop`, `Key_transactiontime_start` and `Key_transactiontime_stop` of datetime datatype, which shall hold the transaction-time period information associated to the `NOT NULL` and `PRIMARY KEY` or `UNIQUE` constraints, respectively.

5.3 Proposed Changes to the MySQL C API

We propose the following changes to the MySQL C API to allow the user to obtain transaction-time and partitioned store related schema information maintained in MySQL.

- The function `mysql_list_tables` returns the list of tables in the database. The signature of this function is given below.

  ```
  MYSQL_RES *mysql_list_tables(MYSQL *mysql, const char *wild)
  ```

  We propose a new function `mysql_list_tables_tt` which shall return the list of tables along with transaction-time and partitioned store related information associated to the tables. The signature of the new function shall be the same as the above except the function name. The new function shall differ from the existing function with respect to the columns returned in the `MYSQL_RES` result set. The columns returned in the result set of the new function shall be the same as those returned when the command `SHOW TRANSACTIONTIME TABLES` is executed. The result set associated to this command has been discussed in Section 5.2.8.2.

- The function `mysql_list_fields` returns the list of fields in a given table along with their attributes. The signature of this function is given below.

  ```
  MYSQL_RES *mysql_list_fields(MYSQL *mysql, const char *table,
                                              const char *wild)
  ```

  We propose a new function `mysql_list_fields_tt` which shall return the transaction-time related information associated to the columns of the table, along with the other attributes of the columns. The signature of the new function shall be same as the above except the function name. The new function shall differ from the existing function with respect to the columns returned in the `MYSQL_RES` result set. The columns returned in the result set of the new function shall be the same as those returned when the command `DESC TRANSACTIONTIME` *table_name* is executed. The result set associated to this command has been discussed in Section 5.2.8.2.

5.4 Summary of Changes Proposed

The changes proposed to MySQL Language in this section aim towards providing transaction-time and partitioned store support for MySQL. The changes allow the creation of tables with transaction-time and partitioned store support. Further, table constraints and column constraints can be created with transaction-time support. The tables can also be altered to add or drop transaction-time or partitioned store support. The changes modify the semantics of insert, update and delete statements such that these statements shall operate on the current transaction-time value of tables which have transaction-time support. The changes also allow sequenced and nonsequenced queries to be performed. The proposal describes the definition and information schema changes that would be needed to maintain the transaction-time and partitioning related information associated with the various database objects.

6 Architecture

τBerkeleyDB and τMySQL are existing systems. In the following sections we explain their existing and extended architecture.

6.1 Existing Architecture

Figure 1 shows the various modules present in the existing architecture. Each rectangle in the figure represents a module and the lines between them represent the interaction between the modules. The dotted rectangles show the modules that comprise τBerkeleyDB and τMySQL. The clock module and the stamper module interact with BerkeleyDB to provide transaction-time support. The TUC module ensures temporal upward compatibility in BerkeleyDB. This module acts as a layer of abstraction between BerkeleyDB and other applications that interact with BerkeleyDB. It exports an interface which comprises of BerkeleyDB API and functions that provide temporal support. This module decides the tasks to be performed based on the temporality of the relation in use. The temporal support module interacts with MySQL to make the temporal features of BerkeleyDB available to the MySQL user. This module interacts with MySQL to allow creation and manipulation of temporal relations.

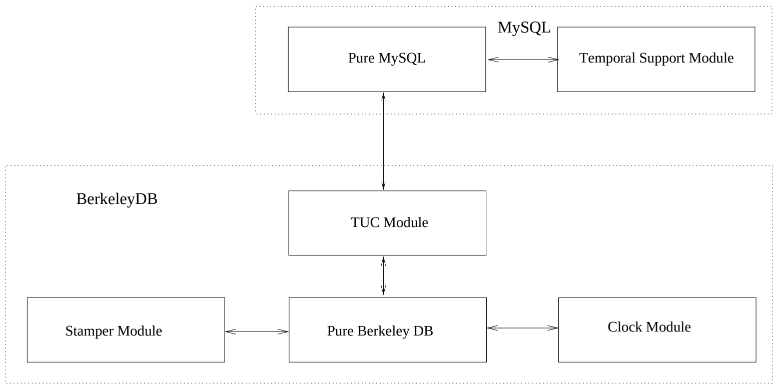

Figure 1: τMySQL-τBerkeleyDB Existing Architecture.

6.2 Extended Architecture

Figure 2 gives a top-level picture of the various modules extended as a part of this thesis. The double boxes in the figure represent modules that have been changed. The TUC module has been modified to manage the storage and retrieval of data from the partitioned store for transaction-time relations. This module abstracts the method of storage (single or partitioned store) and the method of access (Single or Dual B-Tree) from the applications that interact with BerkeleyDB. This module has been modified to decide whether the data should be stored in the current store or archival store and act accordingly. The various decisions made by this module are based on input flags and the temporality of the relation in use.

The MySQL temporal support module has been enhanced to allow the τMySQL user to perform a nonsequenced select. This has been achieved by interacting with MySQL parser to check whether the input query is a nonsequenced select or not and directing the control to the appropriate τBerkeleyDB function. Further, if the input query is nonsequenced then the temporal information obtained from the τBerkeleyDB function is piped through τMySQL so that it is available to the τMySQL user.

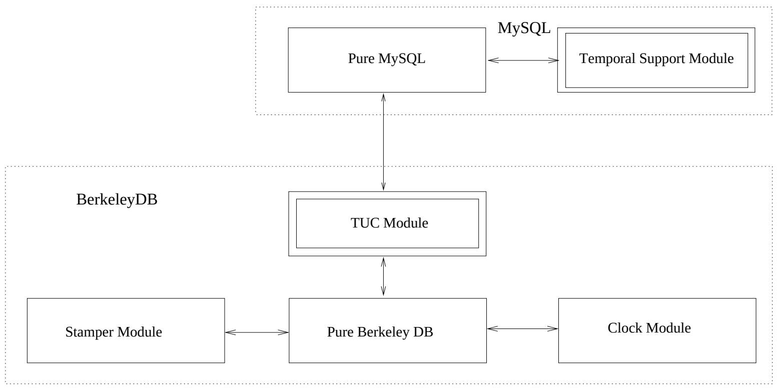

Figure 2: τMySQL-τBerkeleyDB Extended Architecture.

7 Implementation

The following sections provide the implementation details of the various features added by us to τBerkeleyDB and τMySQL.

7.1 Partitioned Store for τBerkeleyDB

The changes done to add partitioned store support to τBerkeleyDB are explained below, based on the four major functions Create, Put, Get, and Nonsequenced get.

- **Create:** This function creates a database based on input parameters. This function has been modified to create an extra database, when it is called to create a transaction-time database with a partitioned store. The extra database is used as the archive store for the database created. The original database created by this function is used as the current store.
- **Put:** This function inserts or updates data in the database. This function has been modified to move the older version of the data into the archive store and to retain the updated/inserted version of the data in the current store. The records in the current store have a start-time associated to them, which represents the transaction-time at which the records acquired their current state.
- **Get:** This function retrieves the current version of all the data matching the input criteria from the database. This function has been modified to return all the matching data present in the current store if the database

has a partitioned store, since the current store contains only the current versions. If the database does not have a partitioned store, then all the data matching the input criteria is scanned, to find the current version and return it, since the single store contains both current and archive records.

- **Nonsequenced get:** This function retrieves current and archive versions of data matching the input criteria from the database. This function has been modified to return the data matching the input criteria from the current store and the archive store if the database has a partitioned store. If the database does not have a partitioned store then all the data matching the input criteria is returned from the single store.

A new function `is_partitioned` was implemented to accept a pointer to a database as the argument and return 1 if that database has a partitioned store, and 0 otherwise. The functions `DB->set_partitioning` and `DB->set_transactiontime` have not been implemented and are part of future work.

7.2 Transaction-Time Support for τMySQL

The available version of τMySQL allowed creation and manipulation of temporal relations. It has been enhanced to allow the τMySQL user to perform a nonsequenced select. It has also been modified to support temporal partitioning, which allows the τMySQL user to specify whether the transaction-time database being created needs to be temporally partitioned or not. The following sections detail the analysis and changes performed on MySQL.

7.2.1 Control Flow of MySQL

In order to modify MySQL to support the features mentioned above the control flow of MySQL was analyzed. The various modules analyzed to understand the control flow have been listed below with a brief description of the associated files. In the following section we describe our changes to these files.

- **User interface module:** This module provides the user with an interactive shell through which the user can execute SQL statements. This module accepts user input, packs the input into structures, sends them to the server, obtains the result from the server, and displays the results to the user. The files associated with this module are under the folder named `client` in the MySQL source tree. The file `client/mysql.cc` contains the driving routine that invokes the other functions in the user interface module. Most of the functions in this module are defined in the file `libmysql/libmysql.c`.

- **Client-server communication module:** This module provides functions which can be used by the client and the server to communicate with each other. The functions in this module act as wrappers to the system calls used for network communication. The files associated to this module are `libmysql/violite.c`, `libmysql/net.c` and `sql/net_serv.cc`.

- **SQL statement parsing module:** This module parses the SQL statement sent by the client and decides the actions that need to be taken. The file `sql/sql_parse.cc` contains the driving routine that invokes the functions which read, parse, process and execute the SQL statement sent by the client, and then return the results to the client. The SQL statement is parsed and validated based on the grammar specified in the `sql/sql_yacc.yy` file. The SQL statement is also validated by checking if the tables and columns referenced by it exist in the database. This module interacts with the table description retrieval module to get the description of the tables and columns referenced by the SQL statement.

- **Query processing module:** This module processes SQL statements which are queries. The file `sql/sql_select.cc` contains the functions that analyze, optimize and process the input query. The functions optimize the query's join order based on the indices available. This module interacts with the data retrieval module to get the data associated to the query being processed.

- **Table description retrieval module:** This module retrieves information about a given table and its columns. The files `sql/sql_base.cc` and `sql/table.cc` contain the functions that retrieve table description by decoding the header information stored in the file associated to the table. This module also caches the table descriptions in order to provide faster response time.

- **Data retrieval module:** This module interacts with the underlying database to retrieve the data from the table being queried. This module abstracts the differences in the interfaces of the various types of underlying databases, viz. BerkeleyDB, InnoDB and ISAM. The `sql/handler.cc` file contains functions that associate the appropriate interface to this module based on the type of the table being accessed. The `sql/ha_berkeley.cc` file contains functions that access the interface provided by BerkeleyDB.

7.2.2 Modifications to MySQL

The modifications done to MySQL have been described below.

- **Query parsing module:** This module parses the user query and decides the actions that need to be taken. This module was modified to set a query-specific flag when the query is nonsequenced.
- **Table description retrieval module:** This module retrieves information about a given table and its columns. This module was modified to return two extra columns "∼*CorrelationName*.STARTTIME" and "∼*CorrelationName*.STOPTIME" of datetime datatype as a part of the table description, if the query is nonsequenced, which is determined by checking whether the query-specific flag is set or not. The two extra columns are setup such that they are the last two columns in the new table description.
- **Data retrieval module:** This module interacts with τBerkeleyDB to retrieve the data from the table being queried. This module was modified to call the `c_get_nonsequenced` function instead of `c_get` function when the query is nonsequenced, so that the current version and the archive version of the data is obtained from the table being queried. Secondly, the STARTTIME and STOPTIME information returned by the `c_get_nonsequenced` function is placed into the MySQL representation of the result record. The MySQL result record contains MySQL-specific flags, record data from BerkeleyDB and a hidden primary key, in the order specified. The STARTTIME and STOPTIME information is placed between the record data from BerkeleyDB and the hidden primary key in the MySQL result record. Since the table description has been adjusted to have two extra columns of datetime datatype, the other modules of MySQL process the STARTTIME and STOPTIME information in the result record in the same manner as they would process any other column of datetime datatype.

7.2.3 Analysis of Changes to MySQL

Out of the changes proposed for the MySQL language enumerated in Section 5, the ones that have been implemented are listed below.

- The words TRANSACTIONTIME and NONSEQUENCED have been added to the list of reserved words in MySQL.
- Changes to the syntax and semantics of <query expression>, so as to provide support for NONSEQUENCED TRANSACTIONTIME SELECT, have been implemented.
- Rules associated to <query specification> have been enforced.
- Changes to the syntax and semantics of <table definition>, so as to support for creation of a transaction-time table with or without a partitioned store have been implemented.

The changes that are yet to be implemented in MySQL, have been listed below.

- Syntax and rules associated to <datetime value function> need to be implemented. The syntax needs to be extended to allow the TRANSACTIONTIME keyword to be specified while invoking functions CURRENT_DATE, CURRENT_TIME or CURRENT_TIMESTAMP. The semantics need to be adjusted to return the datetime information with the appropriate precision. The change can be implemented by following the methodology used to support the existing datetime functions. The `txn_current` function available in the τBerkeleyDB interface has to be invoked and the results obtained should be passed back to the MySQL query processing engine. This change can be implemented in around 1 week (approximately 40 hours).

- Rules associated to <item reference> and <table reference> need to be enforced. The semantics of the <item reference> in the <time option> has to be adjusted to behave like other item references that can exist in the <query expression>. The syntax of the <time option> has to be checked with respect to the scope of the <table reference>, which can be achieved by following the approach used for checking the syntax of the <select list> contained in the <select statement: single row> with respect to the scope of the <table reference>. This change can be implemented in around 1 week.

- Syntax and rules associated to <period value expression> needs to be implemented. The syntax needs to be changed as follows.

  ```
  <transactiontime function> ::=
                      BEGIN <left paren> TRANSACTIONTIME <left paren>
                      <transactiontime argument> <right paren> <right paren>
                 |    END <left paren> TRANSACTIONTIME <left paren>
                      <transactiontime argument> <right paren> <right paren>
  ```

  The rules associated to <transactiontime argument> can be enforced by following the model used to enforce that item qualifiers in a <select list> are references to tables. To implement the semantics, the value of the fields "∼*CorrelationName*.STARTTIME" and "∼*CorrelationName*.STOPTIME" needs to be associated with the respective references to <transactiontime function>. This change can be implemented in around 1 week.

- Implementation of <query expression> syntax and semantics needs to be enhanced to support sequenced queries. The syntax needs to be changed as follows (that is, make NONSEQUENCED optional).

  ```
  <transactiontime option> ::=
              [ NONSEQUENCED ] TRANSACTIONTIME [ <value expression> ]
  ```

  Implementation of the sequenced query semantics would need the implementation of the temporal relational operators. This is a complex task, as it would need modifications to the query evaluation and query processing engine of MySQL.

- <table constraint definition> and <column definition> need to be modified to allow the user to specify transaction-time based constraints. Secondly, the syntax rules associated to <check constraint definition> need to be enforced. The syntax has to be adjusted to allow the specification of <time option> while defining table or column constraints. The semantics have to be adjusted to work on current data or current and archive data based on the <time option> specified. This could be implemented by retrieving the data using `c_get` or `c_get_nonsequenced` based on the <time option> specified. The sequenced semantics associated with this change would need the temporal relational operators to be available. This nonsequenced and current semantics of this change can be implemented in around 3 weeks.

- The <alter table statement> needs to be modified to allow the user to add/drop transaction-time support for a table. The syntax needs to be changed as follows.

  ```
  <alter table action> ::=
              !! All alternatives from ISO/EIC 9075
             | <add transaction definition>
             | <drop transaction definition>
  ```

  Transaction-time support can be added to a database by internally creating a transaction-time database and moving all the records from the original database to it and then dropping the original database. Transaction-time support can be dropped from a database by internally creating a database and moving the current records from the transaction-time database to the new database. The modifications done to <table option> as a part of <table definition> syntax change also affects the syntax of <alter table statement>, so as to allow the user to add/drop partitioned support for a table. The associated syntax is shown below.

```
<alter table action> ::=
        !! Other alternatives from MySQL language
      | <table options>


<table option> ::=
        !! All alternatives from MySQL language
      | PARTITIONED = { 0 | 1 }
```

Adding of partitioned store support would need the existing database store to be split into a current store and an archive store. Dropping of partitioned store support would need the existing current and archive stores to be merged. Since these actions need to be performed without affecting the transaction-time support of the database, the τBerkeleyDB interface would have to support this feature. The function `DB->set_partitioning` discussed in section 4 can be used to achieve this functionality. This change can be implemented in around 2 weeks.

- Syntax and semantics of <select statement: single row> needs to enhanced to support sequenced queries. The syntax needs to be changed as follows.

```
<select statement: single row> ::=
            [ <time option> ]
            SELECT [ <set quantifier> ] <select list>
            INTO <select target list>
            <table expression>
```

This change would need the temporal relational operators to be available.

- Changes to the syntax and semantics of <delete statement: searched> and <update statement: searched> need to be implemented. The syntax needs to be changed as follows.

```
<update statement: searched> ::=
        [ <time option> ]
        UPDATE <table reference>
            SET <set clause list>
            [ WHERE <search condition> ]

<delete statement: searched> ::=
        [ <time option> ]
        DELETE FROM <table reference>
            [ WHERE <search condition> ]
```

Based on the <time option> the where clause of the delete and the update statement needs to be evaluated on the current data or the current and archive data. This could be implemented by retrieving the data using `c_get` or `c_get_nonsequenced` based on the <time option> specified. This change can be implemented in around 4 weeks.

- Transaction-time and partitioned store related information associated to a table needs to be stored as a part of the schema information. This change can be implemented by by adding new members to the `TABLE` and `FIELD` structures present in MySQL, to hold the transaction-time and partitioned store related information. This information can then pushed to the corresponding *table_name*`.frm` file. Secondly, new commands and API need to be provided to return transaction-time and partitioned store related information associated to the tables and columns. The language syntax needs to be changed to allow the commands `SHOW TRANSACTIONTIME TABLES`, `SHOW COLUMNS FROM TRANSACTIONTIME` *table_name* and `DESC TRANSACTIONTIME` *table_name* to be specified by the user. These new commands can be implemented by following the working model of the corresponding MySQL commands. The functions `mysql_list_tables_tt` and `mysql_list_fields_tt`, need to be added to the MySQL C API. These functions can be implemented by executing the corresponding SQL commands and returning the result set obtained from the execution. This change can be implemented in around 4 weeks.

7.3 Summary

This section discusses the changes done to τBerkeleyDB in order to add partitioned store support to it. The changes are explained with respect to the four major functions changed to add this feature, namely Create, Put, Get, and Nonsequenced get. The changes done to τMySQL in order to provide the ability to perform a nonsequenced select are also explained in this chapter. The control flow information of τMySQL, which was collected to perform changes to τMySQL, has been provided. This information would help in performing future changes to τMySQL. The changes that are yet to be implemented in MySQL have been discussed in this chapter, so as to provide an idea of how these changes can be implemented.

8 Experimental Evaluation

The addition of partitioned store support to τBerkeleyDB provides the ability to create a transaction-time database with a partitioned store. The presence of the partitioned store in a transaction-time database is expected to improve the response for current queries. Secondly, in a transaction-time database with a partitioned store, two stores need to be accessed to process update statements and nonsequenced queries, whereas a single store access is enough to process these statements in a transaction-time database without a partitioned store. Hence, the presence of a partitioned store is also expected to affect the performance of update statements and nonsequenced queries. We have designed a set of experiments which will exhibit the effect of adding a partitioned store to a transaction-time database. The following sections describe the experimental setup and the experiments in detail.

8.1 Systems Evaluated

The systems being evaluated by the experiments are the following.

- Berkeley DB loaded with data that would represent the current store of a partitioned Berkeley DB (BDB-C).
- Temporal Berkeley DB loaded with data that would represent the current and archive store of a partitioned Berkeley DB (τBDB).
- Partitioned Temporal Berkeley DB (PBDB).

We also considered the option of having a (conventional) Berkeley DB loaded with data that would represent the current and archive store of a partitioned Berkeley DB (BDBF) as one of the systems for evaluation. Later, BDBF was removed from the list of systems being evaluated, since BDBF very closely represents τBDB with respect to the database size. Secondly, the aim of the experiments is to expose the advantages of PBDB with respect to τBDB, hence evaluation of BDBF is not necessary.

The systems are evaluated based on three metrics: CPU cost, I/O cost and response time. The CPU cost represents the time for which the CPU is used by a particular system for a given task. The I/O cost represents the number of I/O operations done by a particular system for a given task. We do not differentiate between sequential and random I/O. The response time represents the time taken by the system to complete a given task, which represents the sum of CPU cost, I/O cost and system idle time waiting for resources. The response time and CPU cost of a given task is measured by using the time information returned by the `times` function. The I/O module of Berkeley DB was modified to count and report the number of pages accessed while processing a given task. The information returned by this module is used to measure the I/O cost of a given task.

8.2 System Parameters

The system parameters fixed across the various experiments are page size, database size, record size, and the number of records affected in an operation. The values of these parameters are the following.

- Page Size: Default page size of BDB, which is the I/O block size for the file system (8Kb).
- Database Size: 2000 pages.
- Record Size: 250 bytes.

Table 1: Meaning of Variables

S_b	Size of memory buffer pool in pages
N_a	Number of records affected
N_v	Number of versions for every current record in the database
N_c	Number of current records in the database = Total number of records / N_v
N_r	Number of records in a region (discussed below)

- Number of records affected: 320.

The variable parameters in the experiments are the following.

- Type of the statement being executed, which could be current query, temporal query, TUC insert, TUC update or TUC delete.
- Number of versions for every record of data in the database, which takes positive integer values ranging from 1 to 28.
- Size of the buffer pool available to the system for buffering data pages, ranging from 4 pages to 128 pages. In the experiments where the buffer pool size is not varied it is set to 32 pages, which is the default buffer pool size of Berkeley DB.

All the variables used and their meaning has been summarized in Table 1.

8.3 Experimental Setup

The three systems are setup for different values of N_v in the following manner.

- The BDB-C system is setup by creating a BDB database which does not allow duplicates and loading the database with N_c number of records.
- The τBDB system is setup by creating a τBDB database which does not allow duplicates and loading the database with N_c keys and N_v versions for each key. This is achieved by inserting the key/data pair and updating the data $N_v - 1$ times.
- The PBDB system is setup by creating a τBDB database which does not allow duplicates and loading the database with N_c keys and N_v versions for each key. This is achieved by inserting the key/data pair and updating the data $N_v - 1$ times.

Since we expect a linear variation in the behavior of the systems with respect to variations in N_v, we setup the systems for values of N_v as 1, 4, 8, 12, 16, 20, 24, 28. The keys inserted into the systems are of integer data type and range from 1 to N_c for all systems. The associated data values are strings generated by concatenating the key and version information of the corresponding record and padding the result by a default character to extend the length of the string to equal the record size parameter. The key/data pairs are inserted into the systems in a sequential manner, with the pairs ordered by the key value, starting with 1. When the system needs to be setup with N_v versions of a key, the data associated to the key is updated $N_v - 1$ times. The next key in the sequence is inserted into the system only after versions of the previous key are setup in the system. Once the systems are setup, a copy of the associated database files was made, so that it can be reused for each of the experiments.

8.4 Experiments

We perform eight experiments to evaluate the three systems with respect to the variable parameters. The experiments have been grouped based on the type of statement being evaluated in the experiments. The following sections explain the experiments in each group.

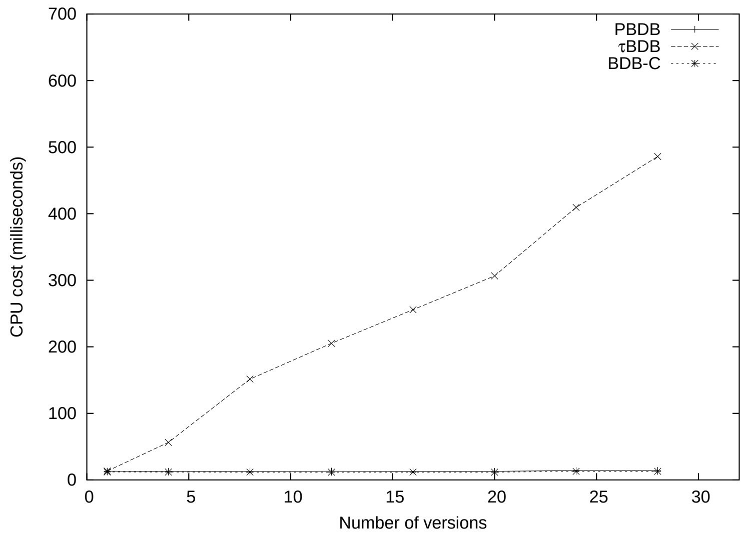

Figure 3: CPU cost vs number of versions for current query.

This figure shows that CPU cost of PBDB is equivalent to that of BDB-C and is invariant with increase in the number of versions for a current query, which is better than the increasing CPU cost observed in τBDB. PBDB has better performance than τBDB in this case since the current query can be answered by accessing the data present in the current store.

8.4.1 Current Query

The PBDB partitions the current data and the archive data, so that it can provide better response for current queries when compared to τBDB. The first experiment in this group compares the CPU cost of the various systems for linearly varying values of N_v with the type of statement being current query. To get an accurate value for the CPU cost, five current queries are executed and the average of these is used as the result. To eliminate any sampling error that might be present in the measurement, the CPU cost is averaged over 1000 runs of the query. In order to ensure that the records affected by one query do not overlap with another, the current queries are set up to query records from different regions in the database. Since five current queries need to be executed, five non-overlapping regions are required. The total number of records in the current store is N_c, so the number of records in a region N_r would be N_c/5. So, we can define the regions as records with key values ranging from 1 to N_r, N_r+1 to 2·N_r, . . . , 4·N_r+1 to 5·N_r. The queries will select the first N_a ($< N_r$) records in the corresponding regions.

Figure 3 shows the results obtained from this experiment. We expect PBDB and BDB-C to be invariant to the change in N_v, since a current query accesses the current store, whereas the change in N_v only affects the archive store. The results are as expected. Secondly, we expect τBDB to have a linear increase in CPU cost with an increase in N_v, since more versions implies more pages to be fetched and processed to get the current records that belong to the result set. We observe that the CPU cost associated to τBDB increases almost linearly with increase in N_v. The slope of this linear increase is approximately 16 milliseconds per version. This experiment helps us infer that PBDB provides better performance for current queries when compared to τBDB, and it does equal the performance of BDB-C.

The second experiment in this group compares the I/O cost of the various systems for linearly varying values of N_v with the type of statement being current query. The queries are setup in the same manner as the first experiment. As there is no sampling error associated to the I/O cost, each query is executed once. The lowest of the I/O cost of the five queries is used as the result, so as to not consider the extra page accesses that could have been due to the placement of the starting key of the query in the middle or end of a page.

Figure 4 shows the results obtained from this experiment. We expect PBDB and BDB-C to be invariant to the change in N_v, since a current query accesses the current store, whereas the change in N_v only affects the archive store. The results for PBDB and BDB-C are as expected. In the case of τBDB, the number of pages accessed is

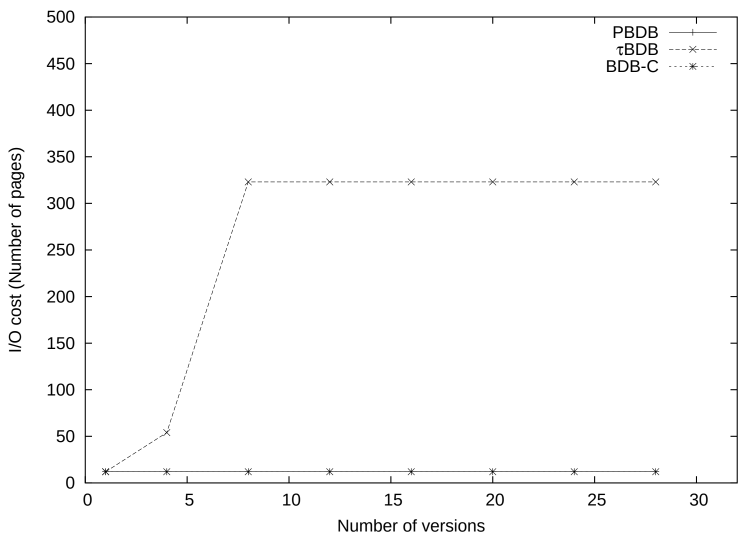

Figure 4: I/O cost vs number of versions for current query.
This figure shows that I/O cost of PBDB is equivalent to that of BDB-C and is invariant with increase the in number of versions for a current query, which is better than the high I/O cost observed in τBDB. PBDB has better performance than τBDB in this case since the current query can be answered by accessing the data present in the current store.

---

expected to be $\lceil N_a \cdot N_v/(\text{Number of records per page})\rceil$, if the pages are full. Considering a case where the number of records per page is 32, if N_v = 1 and N_a = 100 then $\lceil 100{\cdot}1/32 \rceil$ = 4 pages would have to be processed by τBDB; if N_v = 2 and N_a = 100 then $\lceil 100{\cdot}2/32 \rceil$ = 7 pages would have to be processed by τBDB. Since I/O cost is measured based on the number of pages accessed rather than the number of records accessed, the blocking factor of the page is expected to bring about a step effect in the linear increase of I/O cost of τBDB with an increase in N_v. The results obtained are different from what was expected.

From the results, it is observed that there is a sudden increase in the I/O cost of τBDB when N_v = 8. On analyzing this case further, it is found that at N_v = 8 all the pages in the system are split in such a way that each page has only one key and its 8 version records. It is also observed that the I/O cost remains the same as N_v increases from 8 to 28, which can be attributed to the page split at N_v = 8, which would have created enough space in each page for the incoming version records. A test experiment was performed to find out when the next rise would occur in the I/O cost. It was found that at N_v = 30 each of the leaf pages in the system is filled with the first 29 versions of a key and the 30th version is placed in an overflow page, since the leaf pages are full. Hence at N_v = 30 a current query in τBDB needs to access a leaf page and an overflow page for retrieving each current record, which causes the I/O cost to double.

The page split behavior observed at N_v = 8 affects the observations in all the experiments hence we analyzed this case in detail. It was found that the behavior observed at N_v = 8 is a result of the combined effect of the page split criterion used by Berkeley DB, the record size and page size values used, and the method used to setup the systems. Berkeley DB splits a page when the page is filled to an extent greater than a certain threshold value called the fill factor. The record size and page size values decide the number of records that would cause the page to be filled to an extent greater than the fill factor. As mentioned earlier, we setup the systems by inserting the key/data pairs in a sequential manner, with the pairs ordered by the key value. Due to the uniform way in which the systems are setup, all the pages undergo the process of being filled with more than one key initially, crossing the fill factor as the eighth version is inserted and being split to have only one key and its versions, at the same time. This causes the behavior observed by us at N_v = 8.

The third experiment in this group compares the response time of the various systems for linearly varying values of N_v with the type of statement being current query. For this experiment, two queries are setup to fetch $3{\cdot}N_a$ records from two non-overlapping regions, instead of five queries fetching N_a records from five non-overlapping

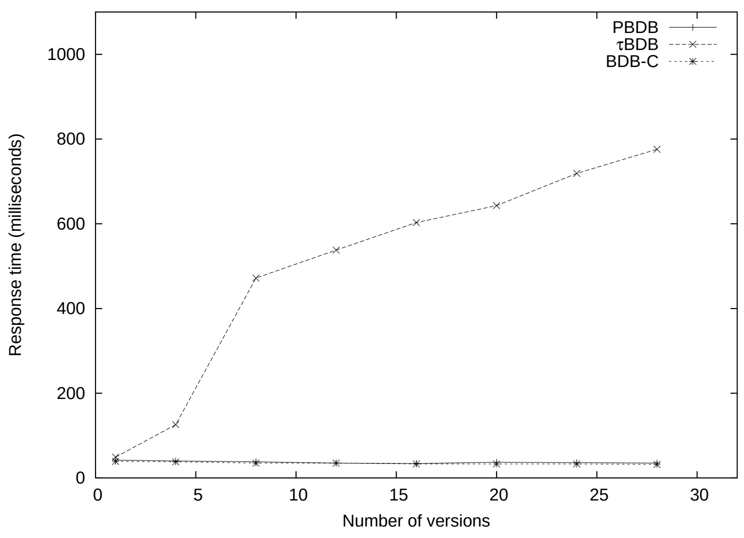

Figure 5: Response time vs number of versions for current query.
This figure shows that the response time represents the cumulative effect of I/O cost and CPU cost.

regions as in the first experiment. This was done in order to have a larger result set and a higher response time, which would help in reducing the sampling error. The experiment is run multiple times on a cold cache and the values are divided by three and averaged over the multiple runs to get the response time for fetching N_a records.

Figure 5 shows the results obtained from this experiment. We expect the results of this experiment to represent an aggregate of the results from the first and the second experiments. We observe a sudden rise in response time at N_v = 8, which is an artifact of the I/O cost. We also observe a linear increase in response time as N_v increases from 8 to 28, which is an artifact of the CPU cost. Hence, the results of this experiment are as expected. They help us understand that response time represents the I/O cost and CPU cost put together, hence its a redundant metric and need not be evaluated in the other experiments.

8.4.2 Update Statement

The overhead faced by PBDB while partitioning the data can be seen by comparing the CPU cost of the various systems for linearly varying values of N_v and the type of statement being an update, which is the first experiment in this group. We do not evaluate BDB-C in this experiment, since N_v does not affect BDB-C. Moreover our aim in this experiment is to look at the overhead of partitioning the database, so comparing PBDB with τBDB is sufficient to accomplish our aim. The update statement is set up in the same manner as the current query statement in the first experiment. The experiment is run multiple times on the system, wherein the system is restored to its original state after each run, since the experiment modifies the data in the system. The average of the values obtained from the various runs and the various regions is used as the result.

Figure 6 shows the results obtained from this experiment. We expect the CPU cost to increase linearly with increase in N_v for PBDB and τBDB. The line representing PBDB is expected to be slightly higher than the line representing τBDB, which indicates the overhead faced by PBDB during update operations. The overhead faced by PBDB during an update is the task of pushing the older version of the updated data to the archive store. We observe that the CPU cost associated to τBDB increases almost linearly with increase in N_v. We observe that the CPU cost associated to PBDB is lesser than τBDB in most of the cases, which is because PBDB does not have to scan through the various versions of data to find the current record and update its time information, as it can find the current record in the current store. We observe a sudden rise in the CPU cost of PBDB for N_v = 8, wherein there is one version in the current store and seven versions in the archive store initially, and the update causes the archive store to have eight versions. This observation can be attributed to the page split behavior discussed in Section 8.4.1.

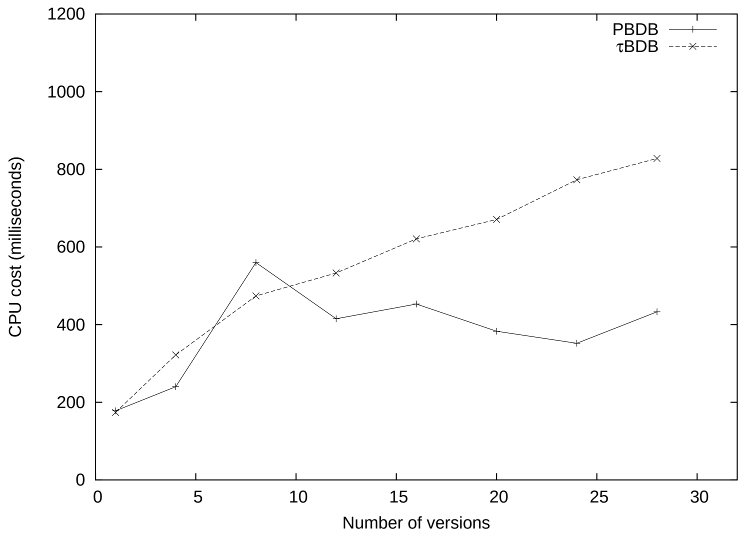

Figure 6: CPU cost vs number of versions for update statement.
This figure shows that CPU cost of τBDB increases linearly with increase in the number of versions for an update statement. PBDB has a lower CPU cost when compared with τBDB as it does does not have to scan through the various versions of data to find the current record, since the current record is present in the current store.

The second experiment in this group compares the I/O cost of PBDB and τBDB for linearly varying values of N_v and the type of statement being update. The updates are setup and the results obtained are evaluated in the same manner as the second experiment. We expect PBDB to have a slightly higher I/O cost than τBDB as it has to write information to both the current store and the archive store in case of an update.

Figure 7 shows the results obtained from this experiment. We observe a sudden rise in the I/O cost of τBDB when N_v = 8, which is due to the page split behavior discussed in Section 8.4.1. We also observe a rise in the I/O cost of PBDB although it has only seven versions in the archive store when N_v = 8, since the update will lead to creation of the eighth version in the archive store. At N_v = 8 the I/O cost of PBDB is lesser than that of τBDB since PBDB reads the pages from the archive store when the number of versions is seven and writes the pages to the archive store with the number of versions being eight. In this case, the number of pages written by PBDB is similar that of τBDB, but the number of pages read is lesser than that of τBDB, since the pages are filled to a greater extent, when the number of versions is seven. For N_v = 12 and above, the I/O cost associated to τBDB and PBDB remain constant, which is due to the initial page split, which would have created enough space in each page for the incoming version records. As expected the I/O cost of PBDB is higher than that of τBDB in most cases, which is due to the overhead of accessing two stores instead of one, both while reading and writing. The constant I/O overhead faced by PBDB is 24 page accesses, which is double the number of pages accessed by PBDB to answer a current query. This observation is valid, since an update operation needs to access the pages twice, once for reading and once for writing.

The delete operation is similar to the update operation in the context of temporal databases. In τBDB, both update and delete operations cause the stop time of the current record to be updated. An update operation also causes an insert of a new record with the new data. Hence, in τBDB, we expect the I/O cost for update and delete to be equivalent and the CPU cost for update and delete to differ by a constant value. In PBDB, both update and delete operations cause a copy of the current record to be moved to the archive store, after updating the stop time appropriately. Further, an update causes the record in the current store to be updated with the new information and a delete causes the record in the current store to be physically deleted. Hence, in PBDB, we expect the I/O cost and CPU cost for update and delete to be equivalent. So, we do not perform another set of experiments to compare the systems with respect to the delete operation. In order to confirm our expectation on the similarity of update and delete operation a test experiment with delete operations for a particular value of N_v was performed. It was found that the observation was correct, except in a case where the update in τBDB accesses more pages than the

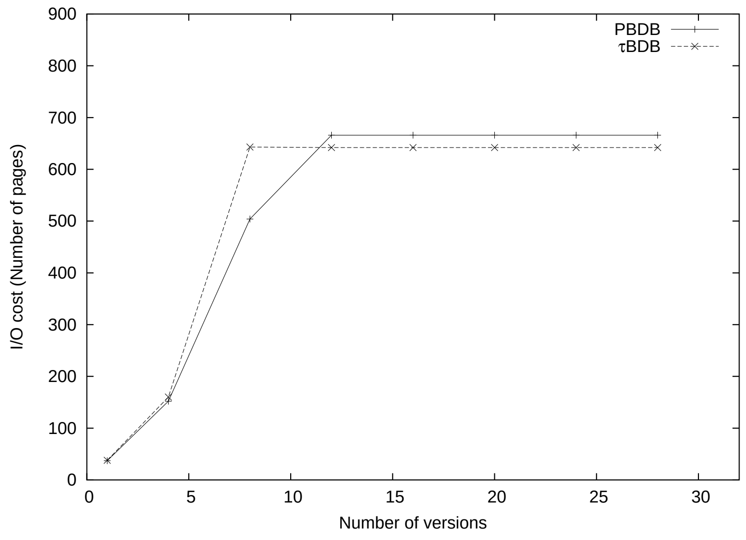

Figure 7: I/O cost vs number of versions for update statement.
This figure shows that, for an update statement the I/O cost of PBDB is higher than that of τBDB in most cases, which is due to the overhead of accessing two stores instead of one, both while reading and writing.

---

delete, if the update is on a page which is already full and the addition of a new version would need the data to be distributed between the current page and a new page.

8.4.3 Nonsequenced Query

A feature provided by the temporal database is the ability to view the version history of the data. To show that this operation in PBDB does not have a significant overhead when compared to τBDB, we compare the CPU cost of the two systems for varying values of N_v with the type of statement being nonsequenced query, as the first experiment in this group. The nonsequenced query is set up in the same way as the current query in the first experiment. To eliminate any sampling error that might be present in the measurement, the CPU cost is averaged over 1000 runs of the query. N_a is given by the product of the number of keys in the query range and N_v. So, the number of keys in the query range needs to be adjusted based on the value of N_v so as to keep N_a constant. We expect the CPU cost to be invariant with respect to change in N_v since N_a is a constant and the type of statement is such that the records affected are present next to each other. Since PBDB has to examine both the archive store and the current store, there will be an overhead, due to which we expect the line representing PBDB to be higher than the line representing τBDB.

Figure 8 shows the results obtained from this experiment. We observe that the CPU cost associated to τBDB has a sudden rise when the N_v = 8 and then reduces gradually. This is due to the page split behavior discussed in Section 8.4.1. The page split causes the pages to be partially filled, due to which there is a need to access more pages to fetch N_a records. As the number of versions increases the pages have more records than earlier and hence the number of pages to be accessed to fetch N_a records decreases. PBDB does not experience this sudden rise since the archive store does not have exactly 8 versions in any run, when N_v = 8, the archive store has seven versions and hence is moderately filled, when N_v = 12 the archive store has 11 versions and would be filled moderately after undergoing a split when it crossed 8 versions. For N_v = 12 and above, the CPU cost associated to τBDB and PBDB vary in a similar manner. As expected the CPU cost of PBDB is higher than that of τBDB due to the overhead of examining two stores instead of one.

The seventh experiment compares I/O cost of τBDB and PBDB for linearly varying values of N_v with the type of statement being nonsequenced query. The nonsequenced queries are setup and the results obtained are evaluated in the same manner as the second experiment. We expect the I/O cost to be invariant with respect to change in N_v since N_a is a constant.

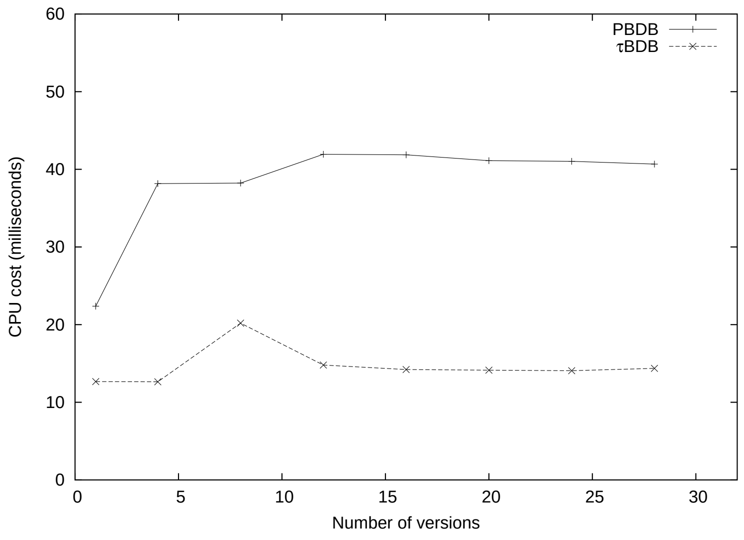

Figure 8: CPU cost vs number of versions for nonsequenced query.
This figure shows that CPU cost of PBDB is higher than that of BDB-C for a nonsequenced query due to the overhead of examining two stores instead of one. The variations in cost are due to the changes in the extent to which the pages are filled for the various values of the number of versions.

Figure 9 shows the results obtained from this experiment. We observe that there is an initial increase and then a decrease in the I/O cost as the number of versions increases. This behavior is due to variation in the extent to which the pages are filled as N_v changes. The pages are least filled when the number of versions for a key is 8 in a particular store, due to the page split behavior discussed in Section 8.4.1. For N_v = 12 and above, the I/O cost associated to τBDB and PBDB vary in a similar manner. As expected the I/O cost of PBDB is slightly higher than that of τBDB due to the overhead of examining two stores instead of one.

8.4.4 Repeated Current Query

When records are accessed repeatedly, the presence of the pages in the memory buffer pool improves the performance. While performing a current query, PBDB would have to get only the pages containing the current version into the buffer pool, whereas τBDB would have to get the pages containing the current version and the history versions, since it does not store them separately. So, to get a certain number of current versions τBDB would have to keep more pages in the buffer pool than PBDB. This can lead to a loss of performance. The only experiment in this group compares I/O cost of τBDB and PBDB for linearly varying values of S_b and a fixed value of N_v. The values of S_b are 4, 8, 12, 16, 24, 32, 64, 96 and 128 pages. We fix N_v to be 4. The query is be to get N_a current records thrice. In τBDB the query will cause $N_a \cdot N_v$ records to be accessed. In PBDB the query will cause N_a records to be accessed. At low values of S_b we expect both PBDB and τBDB to have a high I/O cost, as the pages have to be repeatedly fetched from the disk, since there is not enough buffer pool space to hold onto them. At medium values of S_b we expect PBDB to have lesser I/O cost compared to its I/O cost for low values of S_b, since the number of pages repeatedly accessed by it is small and medium buffer pool space would be enough to keep them in memory. At high values of S_b we expect τBDB to have lesser I/O cost compared to its I/O cost for medium values of S_b, since the high buffer pool space would be enough to keep the pages needed by τBDB in memory. Figure 10 shows the results obtained from this experiment. The results are as expected. We observe that the I/O cost of both systems reduces when S_b becomes higher than the number of pages repeatedly accessed by the system.

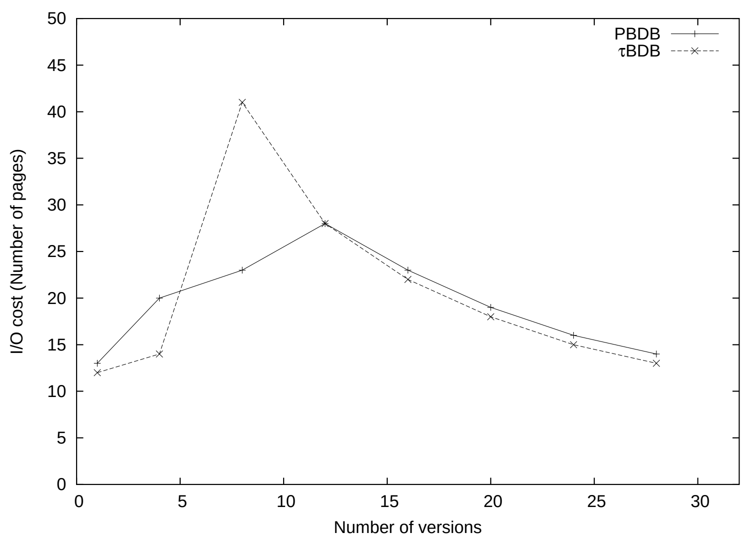

Figure 9: I/O cost vs number of versions for nonsequenced query.
This figure shows that in most cases the I/O cost of PBDB is slightly higher than that of BDB-C for a nonsequenced query due to the overhead of examining two stores instead of one. The variations in cost are due to the changes in the extent to which the pages are filled for the various values of the number of versions.

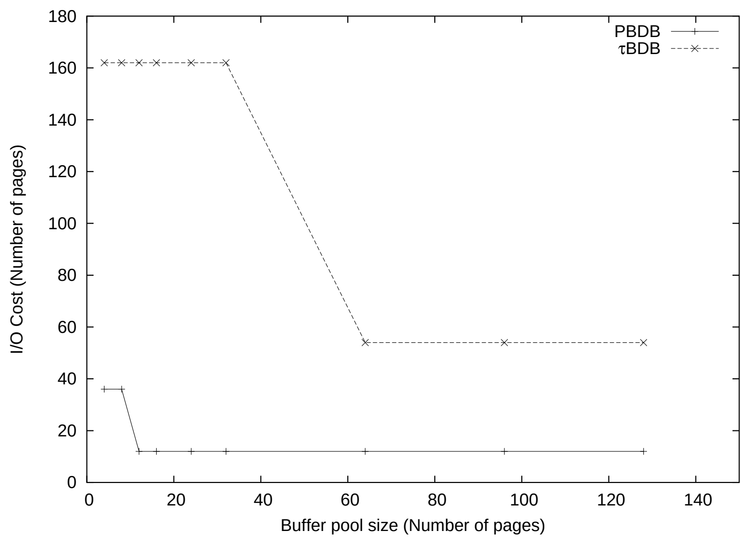

Figure 10: I/O cost vs buffer pool size for repeating current query.
This figure shows that for repeating queries τBDB would need a larger buffer pool than PBDB in order to retain the pages in memory and reduce I/O accesses.

8.5 Summary

The experiments show us the effect of adding a partitioned store to a transaction-time database. We can infer from the first group of experiments that PBDB equals the performance of BDB-C, and provides a much better performance than τBDB with respect to current queries. The second group of experiments suggest that PBDB can provide better performance than τBDB with respect to update statements if the number of versions is high, since I/O overhead faced by PBDB is constant with increase in the number of versions, whereas the CPU overhead faced by τBDB increases linearly with increase in the number of versions. The third group of experiments show that τBDB provides better performance than PBDB with respect to nonsequenced queries. The experiment in the fourth group shows that τBDB needs a larger memory buffer pool than PBDB to answer a repeating current query without making repeated I/O accesses.

9 Conclusion and Future Work

Transaction-time databases provide the ability to maintain the history of the data as it evolves. Conventional database users would transition to transaction-time databases, if the impact of the transition is low. We have implemented a temporally partitioned store in the transaction-time database τBerkeleyDB so as to retain the performance of current queries between conventional databases and transaction-time databases, so that a transition from the former to the latter has a low impact on current queries. We have performed experiments to evaluate the effect of the transition on the performance of current queries and shown that the performance of current queries is comparable in both databases. The experiments also show that the addition of partitioned store support to a transaction-time database adds negligible overhead to the insert, update and delete operations.

We have proposed changes to MySQL language to provide transaction-time support. We have also analyzed the possible ways in which these changes can be applied to τMySQL. We have modified τMySQL to support nonsequenced select and a temporally partitioned store. We also provide information about the working of the various modules in MySQL, which were analyzed by us in order to understand the control flow of MySQL. This information would help in implementing the remaining changes.

τBerkeleyDB and τMySQL can be further enhanced to provide a better system and more features. The following are some of the areas that should be investigated.

- In our partitioned store model only current records are kept in the current store so we can reduce the time information stored with these records, as we would not have to store the stop time for these records as all of them will have a unknown stop time until they are changed or deleted. Once they are changed or deleted they will be moved to the archive store, where the stop time can be associated to them. Not storing the stop time in these records will reduce the record size, which will allow more records to fit in a page, which would lead to better performance for queries.

- The store of a transaction-time database and the current and the archive store of a partitioned transaction-time database in τBerkeleyDB are indexed using the B-Tree based access method available in BerkeleyDB. As this is a key-only access method, it would provide an data access path for transaction pure-key and transaction range-timeslice queries, whereas transaction pure-timeslice queries would have to scan the complete database. Transaction pure-timeslice queries can also be provided a data access path by having a time-key access method. Such an access method can be provided for the transaction-time database or for the archive store of a partitioned transaction-time database. The performance implications of such an index should be investigated.

- In τBerkeleyDB the decisions of whether a database needs transaction-time support or not and whether it needs a partitioned store or not have to be made by the user while creating the database. Once created, these options cannot be changed for a database. Allowing changes to these options to be performed at any time would provide more flexibility to the user.

- In a non-partitioned transaction-time database in τBerkeleyDB the versions of a particular key are stored such that the current version is the last one to be fetched while fetching all the versions. This affects the response to current queries. The response to current queries could be improved by storing the versions of a key such that the current version is the first one to be fetched.

- A transaction-time database keeps growing in size as the data undergoes changes. The increase in size can lead to a reduction in performance. Some applications might not be interested in keeping version information which is older than a certain period. To satisfy such a requirement the database needs to be enhanced to provide the database administrator with the ability to either delete or move to another store, version information which is older than a certain period. This process could also be automated by providing the ability to set a threshold period per table, wherein any version record in the table which is older than this threshold will be automatically moved to another store or physically deleted.
- The section 7.2.3 mentions the list of changes that are yet to be applied to the MySQL language. These can be implemented to provide complete transaction-time support in τMySQL.

The implementation of a partitioned store in τBerkeleyDB and the modification of τMySQL to support non-sequenced select are steps towards the goal of making transaction-time databases more available and usable. The proposed future work also works toward achieving this goal.

References

[1] τberkeleydb. `http://www.cs.arizona.edu/tau/tbdb/`.

[2] B. Salzberg and V. J. Tsotras. Comparison of access methods for time-evolving data. *ACM Computing Surveys*, 31(2):158–221, June 1999.

[3] I. Ahn and R.T. Snodgrass. Partitioned storage for temporal databases. *Information Systems*, 13(4):369–391, May 1988.

[4] J. Melton. SQL/Temporal. *ISO/IEC JTC 1/SC 21/WG 3 DBL-MCI-012*, July 1996.

[5] K. Torp, C. S. Jensen and M. H. Böhlen. Layered temporal DBMS's—concepts and techniques. In *Proceedings of the Fifth International Conference on Database Systems for Advanced Applications*, pages 371–380, Melbourne, Australia, April 1997.

[6] K. Torp, C. S. Jensen and R. T. Snodgrass. Stratum approaches to temporal DBMS implementation. In *Proceedings of the International Database Engineering and Applications Symposium*, pages 4–13, Cardiff, Wales, U.K., July 1998.

[7] R. T. Snodgrass. Addendum to valid- and transaction-time proposals. *ANSI X3H2-96-582, ISO/IEC JTC 1/SC 21/WG 3 DBL MAD-203*, December 1996.

[8] R. T. Snodgrass and I. Ahn. A taxonomy of time in databases. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 236–246, New York, NY, May 1985. ACM Press.

[9] R. T. Snodgrass and I. Ahn. Temporal databases. *IEEE Computer*, 19(9):35–41, September 1986.

[10] R. T. Snodgrass, M. H. Böhlen, C. S. Jensen and A. Steiner. Adding transaction time to SQL/Temporal. *ANSI X3H2-96-502r2, ISO/IEC JTC 1/SC 21/WG 3 DBL MAD-147r2*, November 1996.

[11] R. T. Snodgrass, M. H. Böhlen, C. S. Jensen and A. Steiner. Adding valid time to SQL/Temporal. *ANSI X3H2-96-501r2, ISO/IEC JTC 1/SC 21/WG 3 DBL-MAD-146r2*, November 1996.

Appendix

A Examples

This following sections give examples of the various features of τBerkeleyDB and τMySQL, which were discussed in the document.

A.1 τBerkeleyDB Examples

The following is a list of examples demonstrating the various features of τBerkeleyDB.

- **Creating a database in τBerkeleyDB:** The following code snippet gives the sample code showing how the existing interface of BerkeleyDB has been reused to provide the ability to create either a conventional database or a transaction-time database or a transaction-time database with a partitioned store, by just changing the flags passed to the `DB->open` function call.

```
/* creating a conventional database */
ret = db->open(db, txnid, "cbdb.db", NULL,
                   DB_BTREE, DB_CREATE, fileMode);

/* creating a transaction-time database */
ret = db->open(db, txnid, "tbdb.db", NULL, DB_BTREE,
                   DB_CREATE|DB_TEMPORAL, fileMode);

/* creating a transaction-time database with a partitioned store */
ret = db->open(db, txnid, "pbdb.db", NULL, DB_BTREE,
                   DB_CREATE|DB_TEMPORAL|DB_PARTITION, fileMode);
```

- **TUC in τBerkeleyDB:** TUC in τBerkeleyDB ensures that each non-temporal query will return the same result on an equivalent snapshot database as on the temporal counterpart of the database. For example, if we have three databases `emp` (employee database), `empT` (employee database with transaction-time support), and `empTP` (employee database with transaction-time support and a partitioned store), the code snippet shown below will return the same set of records when executed with either `emp` or `empT` or `empTP` as the input database.

```
int displayDB(char * dbName, int openFlag, int numRecords) {

  DB *dbp;
  DBC *dbcp;
  DBT key, data;
  int ret;
  DB_ENV *dbenv;
  DB_TXN * tid;
  /* open environment, database and transaction */
  if ((ret = dbStartup(dbName, &dbp, &dbenv, &tid, openFlag)) != 0)
    return ret;
  /* Acquire a cursor for the database. */
  if ((ret = dbp->cursor(dbp, tid, &dbcp, 0)) != 0) {
    dbp->err(dbp, ret, "DB->cursor");
    return(-1);
  }
  /* Initialize the key/data return pair. */
  memset(&key, 0, sizeof(key));
  memset(&data, 0, sizeof(data));
  /* Walk through the database and print out the key/data pairs. */
  while ((ret = dbcp->c_get(dbcp, &key, &data, DB_NEXT)) == 0
                                  && numRecords-- > 0) {
    printf("(%d, %.*s)\n", *(int *)key.data, (int)data.size,
                                           (char *)data.data);
  }
  if (ret != DB_NOTFOUND && ret != 0) {
    dbp->err(dbp, ret, "DBcursor->get");
    return(-1);
  }
  /* close cursor */
  if ((ret = dbcp->c_close(dbcp)) != 0) {
    dbp->err(dbp, ret, "db cursor close");
  }
  /* close environment, database and transaction */
  if ((ret = dbShutdown(dbp, dbenv, tid, FALSE)) != 0)
    return ret;
  return (0);
}
```

TUC in τBerkeleyDB also ensures that the interface to insert, update and delete data remain the same. The following code snippet will have the same effect on all the three databases `emp`, `empT` and `empTP` and can be executed on all the three databases without any code change.

```
int insOrUpdData(DB *dbp, DB_TXN *tid, int keyVal, char * strData) {

  DBT key, data;
  int ret;
  /* Initialize Key and Data */
  memset(&key, 0, sizeof(key));
  memset(&data, 0, sizeof(data));
  key.data = (int *) &keyVal;
  key.size = sizeof(keyVal);
  data.data = strData;
  data.size = strlen(strData)+1;
  /* Insert/Update the data in the database */
  if ((ret = dbp->put(dbp, tid, &key, &data, 0)) == 0)
    printf(" Stored: (%d, %s) \n", *(int *)key.data,
                                (char *)data.data);
  else {
    dbp->err(dbp, ret, "DB->put");
    return(-1);
  }
  return(0);
}

int deleteKey(DB *dbp, DB_TXN * tid, int keyVal) {

  DBT key;
  int ret;
  /* Initialize Key */
  memset(&key, 0, sizeof(key));
  key.data = (int *) &keyVal;
  key.size = sizeof(keyVal);
  /* Delete Data */
  if ((ret = dbp->del(dbp, tid, &key, 0)) == 0)
    printf(" %d: key was deleted.\n", *(int *)key.data);
  else {
    dbp->err(dbp, ret, "DB->del");
    return(-1);
  }
  return(0);
}
```

- **Nonsequenced get in τBerkeleyDB:** The nonsequenced get in τBerkeleyDB retrieves current and archive versions of data matching the input criteria, from the database. The code snippet shown below retrieves all the records in a given database along with their version history.

```
int displayNonSeqDB(char * dbName, int openFlag) {

  DB *dbp;
  DBC *dbcp;
  DBT key, data;
  int ret;
  struct timeval start, stop, beginning, forever, prevStop, prevStart;
  DB_ENV *dbenv;
  DB_TXN * tid;
  /* open environment, database and transaction */
  if ((ret = dbStartup(dbName, &dbp, &dbenv, &tid, openFlag)) != 0)
    return ret;
  /* Acquire a cursor for the database. */
  if ((ret = dbp->cursor(dbp, tid, &dbcp, 0)) != 0) {
    dbp->err(dbp, ret, "DB->cursor");
    return(-1);
  }
  /* Initialize the key/data return pair. */
  memset(&key, 0, sizeof(key));
  memset(&data, 0, sizeof(data));
  /* Walk through the database and print out the key/data pairs. */
  beginning.tv_sec = BEGINNING;
  beginning.tv_usec = 0;
  forever.tv_sec = FOREVER;
  forever.tv_usec = 0;
  prevStop.tv_sec = 0;
  prevStop.tv_usec = 0;
  prevStart.tv_sec = 0;
  prevStart.tv_usec = 0;
  while ((ret = dbcp->c_get_nonsequenced(dbcp, &key, &data, beginning,
                          forever, DB_NEXT, &start, &stop)) == 0) {
    printf(" (%d, %.*s) start-time: (((%d))), stop-time: (((%d)))\n",
                 *(int *)key.data, (int)data.size, (char *)data.data,
                 start.tv_sec, stop.tv_sec);
  }
  if (ret != DB_NOTFOUND && ret != 0) {
    dbp->err(dbp, ret, "DBcursor->get");
    return(-1);
  }
  /* close cursor */
  if ((ret = dbcp->c_close(dbcp)) != 0) {
    dbp->err(dbp, ret, "db cursor close");
  }
  /* close environment, database and transaction */
  if ((ret = dbShutdown(dbp, dbenv, tid)) != 0) return ret;
  return (0);
}
```

A.2 τMySQL Examples

The following is a list of examples demonstrating the various features of τMySQL.

- **Creating a transaction-time table in τMySQL:** A transaction-time table can be created in τMySQL using τBerkeleyDB as the underlying database. The table can be created with or without a partitioned store by setting `PARTITIONED` to be 1 or 0. The following is an example creating the employee table with transaction-time support and a partitioned store.

```
CREATE TABLE employee(eid INTEGER PRIMARY KEY,
               ename VARCHAR(12), salary INTEGER)
AS TRANSACTIONTIME
TYPE = BDB
PARTITIONED = 1
```

- **TUC in τMySQL:** TUC in τMySQL ensures that each non-temporal query will return the same result on an equivalent snapshot table as on the temporal counterpart of the table. For example, if we have three tables `emp` (employee table), `empT` (employee table with transaction-time support), `empTP` (employee table with transaction-time support and a partitioned store), then the following queries would return the same result.

```
SELECT * FROM emp;
SELECT * FROM empT;
SELECT * FROM empTP;
```

If the employees are Franziska and Therese with employee ID 1 and 2 and salary 6000 and 4000, then the result set would be as shown below.

```
+------+-----------+---------+
| eid  | ename     | salary  |
+------+-----------+---------+
|    1 | Franziska |    6000 |
|    2 | Therese   |    4000 |
+------+-----------+---------+
```

TUC in τMySQL also ensures that update, delete and insert operations work consistently on all the three tables without any change of syntax. For example, the following statements will produce the same effect on all the three tables, where the *tableName* is substituted by `emp`, `empT` or `empTP`.

```
UPDATE tableName
  SET salary = 5000
  WHERE eid = 2;

DELETE FROM tableName
  WHERE eid = 1;

INSERT INTO tableName
  VALUES (3, 'Lilian', 4500);
```

Hence, after these operations are performed the result obtained by executing the queries discussed earlier will also be the same. The result set would be as shown below.

```
+------+---------+---------+
| eid  | ename   | salary  |
+------+---------+---------+
|    2 | Therese |    5000 |
|    2 | Lilian  |    4500 |
+------+---------+---------+
```

- **Nonsequenced select in τMySQL:** The nonsequenced select in τMySQL allows the user to view current and archive versions of data from tables in the database. The query shown below retrieves the employee information from the employee table with the version history.

  ```
  NONSEQUENCED TRANSACTIONTIME
  SELECT  ename, salary,
          BEGIN(TRANSACTIONTIME(employee)) as startTime,
          END(TRANSACTIONTIME(employee)) as stopTime
  FROM employee
  ```

  This query evaluates to the following, wherein the employee record of Franziska was created on 1995-01-01 and updated on 1995-02-01, the employee record of Lilian was created on 1995-01-01 and deleted on 1995-03-02 and the employee record of Therese was inserted on 1995-02-01.

  ```
  +-----------+--------+---------------------+---------------------+
  | ename     | salary |      startTime      |      stopTime       |
  +-----------+--------+---------------------+---------------------+
  | Franziska |   5000 | 1995-01-01 09:30:30 | 1995-02-01 10:30:24 |
  | Franziska |   6000 | 1995-02-01 10:30:24 | 9999-12-31 00:00:00 |
  | Therese   |   4000 | 1995-02-01 10:31:45 | 9999-12-31 00:00:00 |
  | Lilian    |   4500 | 1995-02-02 09:45:12 | 1995-03-02 10:12:22 |
  +-----------+--------+---------------------+---------------------+
  ```

  Note: This feature has not yet been implemented.

- **Retrieving table description in τMySQL:** The command `SHOW TABLES` returns the list of tables in the current database. For example, executing this command on a database named `test` which has three tables `emp`, `empT` and `empTP` will return the following result set.

  ```
  +----------------+
  | Tables_in_test |
  +----------------+
  | emp            |
  | empT           |
  | empTP          |
  +----------------+
  ```

- **Retrieving transaction-time table description τMySQL:** The command `SHOW TRANSACTIONTIME TABLES` shall return the list the tables in the database along with transaction-time and partitioned store related information associated to the tables. For example, executing this command on a database named `test` which has three tables `emp`, `empT` and `empTP` will return the following result set.

```
+----------------+----------------+-------------+
| Tables_in_test | Transactiontime | Partitioned |
+----------------+----------------+-------------+
| emp            |                |             |
| empT           | YES            |             |
| empTP          | YES            | YES         |
+----------------+----------------+-------------+
```

- **Retrieving column description in τMySQL:** The commands `SHOWCOLUMNS FROM` *table_name* and `DESC` *table_name* return the list of columns in the table specified along with their attributes. For example, executing either of these commands with the *table_name* as `emp` will return the following result set.

```
+--------+-------------+------+-----+---------+-------+
| Field  | Type        | Null | Key | Default | Extra |
+--------+-------------+------+-----+---------+-------+
| eid    | int(11)     |      | PRI | 0       |       |
| ename  | varchar(12) | YES  |     | NULL    |       |
| salary | int(11)     | YES  |     | NULL    |       |
+--------+-------------+------+-----+---------+-------+
```

- **Retrieving transaction-time column description in τMySQL:** The commands `SHOW COLUMNS FROM TRANSACTIONTIME` *table_name* and `DESC TRANSACTIONTIME` *table_name* shall show the transaction-time related information associated to the columns of the table, along with the other attributes of the columns. Consider `emp` as a table with four columns `eid`, `ename`, `deptno` and `sid`, where the column `eid` has a `PRIMARY KEY` constraint associated to it, the column `ename` has a `SEQUENCED UNIQUE` constraint with time period 1995-01-01 to 1999-01-01 associated to it, the column `deptno` has a `SEQUENCED NOT NULL` constraint with time period 1995-01-01 to 1999-01-01 associated to it, and the column `sid` has `NONSEQUENCED UNIQUE` constraint associated to it. In such a case executing either of the above commands with the *table_name* as `emp` will return the following result set.

```
+--------+--------------+------+-----+---------+-------+---
| Field  | Type         | Null | Key | Default | Extra |
+--------+--------------+------+-----+---------+-------+---
| eid    | int(11)      |      | PRI | 0       |       |...
| ename  | varchar(12)  | YES  | MUL | NULL    |       |...
| deptno | int(11)      |      |     | 0       |       |...
| sid    | int(11)      | YES  | MUL | NULL    |       |...
+--------+--------------+------+-----+---------+-------+---
---+----------------------+---------------------+---
   | Null_transactiontime | Key_transactiontime |
---+----------------------+---------------------+---
...|                      |                     |...
...|                      | SEQUENCED           |...
...| SEQUENCED            |                     |...
...|                      | NONSEQUENCED        |...
---+----------------------+---------------------+---
---+----------------------------+---------------------------+---
   | Null_transactiontime_start | Null_transactiontime_stop |
---+----------------------------+---------------------------+---
...|                            |                           |...
...|                            |                           |...
...| 1995-01-01 00:00:00        | 1999-01-01 00:00:00       |...
...|                            |                           |...
---+----------------------------+---------------------------+---
---+---------------------------+--------------------------+
   | Key_transactiontime_start | Key_transactiontime_stop |
---+---------------------------+--------------------------+
...|                           |                          |
...| 1995-01-01 00:00:00       | 1999-01-01 00:00:00      |
...|                           |                          |
...|                           |                          |
---+---------------------------+--------------------------+
```