Temporal features in SQL:2011

Krishna Kulkarni, Jan-Eike Michels (IBM Corporation)
{krishnak, janeike}@us.ibm.com

ABSTRACT

SQL:2011 was published in December of 2011, replacing SQL:2008 as the most recent revision of the SQL standard. This paper covers the most important new functionality that is part of SQL:2011: the ability to create and manipulate temporal tables.

1. Introduction

SQL is the predominant database query language standard published jointly by ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission). In December 2011, ISO/IEC published the latest edition of the SQL standard, SQL:2011. A recent article in *SIGMOD Record* provides a brief survey of the new features in SQL:2011 [1]. Because of space constraints, it did not cover the most important new feature in SQL:2011: the ability to create and manipulate temporal tables, i.e., tables whose rows are associated with one or more temporal periods. This is the subject of the current article.

2. Temporal data support

Extensions to support temporal data[1] in SQL have long been desired. There is a large body of research papers, conference publications, and books on this topic, some dating back to the early 1980s. For more details, we refer the readers to an extensive (but outdated) bibliography [2] and to books such as [3] and [4].

Though the previous academic research produced a large number of solutions, the commercial adoption has been rather slow. It is only recently that commercial database management systems (DBMSs) have begun to offer SQL extensions for managing temporal data [5, 6, 7]. Prior to this development, users were forced to implement temporal support as part of the application logic, which often resulted in expensive development cycles and complex, hard-to-maintain code.

In 1995, the ISO SQL committee initiated a project to create a new part of SQL standard devoted to the language extensions for the temporal data support. A set of language extensions based on (but not identical to) TSQL2 [8] were submitted for standardization at that time. Unfortunately, these proposals generated considerable controversy (see [9] for more details), and failed to get adequate support from the ISO SQL committee's membership. In addition, there was no indication that key DBMS vendors were planning to implement these extensions in their products. Eventually, the work on this new part was cancelled in 2001.

Recently, a new set of language extensions for temporal data support were submitted to and accepted by the ISO SQL committee. These language extensions are now part of SQL:2011 Part 2, SQL/Foundation [10], instead of appearing as a new part. There is currently at least one commercial implementation [5] based on these extensions that the authors are aware of.

2.1 Periods

The cornerstone of temporal data support in SQL:2011 is the ability to define and associate time periods with the rows of a table. Essentially, a time period is a mathematical interval on the timeline, demarcated by a start time and an end time.

Many treatments of temporal databases introduce a period data type, defined as an ordered pair of two datetime values, for the purpose of associating time periods with the rows of a table. SQL:2011 has not taken this route. Adding a new data type to the SQL standard (or to an SQL product) is a costly venture because of the need to support the new data type in the tools and other software packages that form the ecosystem surrounding SQL. For example, if a period type were added to SQL, then it would have to also be added to the stored procedure language, to all database APIs such as JDBC, ODBC, and .NET, as well as to the surrounding technology such as ETL products, replication solutions, and others. There must also be some means of communicating period values to host languages that do not support period as a native data type, such as C or Java. These factors can potentially slow down the adoption of new type for a long time.

1. Note that there is no single, commonly accepted definition of the term "temporal data". For the purposes of this article, we define "temporal data" to mean any data with one or more associated time periods during which that data is deemed to be effective or valid along some time dimension.

Instead of adding a period type, SQL:2011 adds *period definitions* as metadata to tables. A period definition is a named table component, identifying a pair of columns that capture the period start and the period end time. `CREATE TABLE` and `ALTER TABLE` statements are enhanced with syntax to create or destroy period definitions. The period start and end columns are conventional columns, with separate names. The period name occupies the same name space as column names, i.e., a period cannot have the same name as a column.

SQL:2011 has adopted a closed-open period model, i.e., a period represents all times starting from and including the start time, continuing to but excluding the end time. For a given row, the period end time must be greater than its period start time; in fact, declaring a period definition in a table implies a table constraint that enforces this property.

The literature on temporal databases recognizes two dimensions of time for temporal data support, e.g., see [3]:

• *valid time*, the time period during which a row is regarded as correctly reflecting reality by the user of the database.

• *transaction time*, the time period during which a row is committed to or recorded in the database.

For any given row, its transaction time may arbitrarily differ from its valid time. For example, in an insurance database, information about a policy may get inserted much before that policy comes into effect.

In SQL:2011, transaction time support is provided by system-versioned tables, which in turn contain the *system-time period*, and valid time support is provided by tables containing an *application-time period*[2]. The name of the system-time period is specified by the standard as `SYSTEM_TIME`. The name of an application-time period can be any user-defined name. Users are allowed to define at most one application-time period and at most one system-time period per table.

One of the advantages of the SQL:2011 approach over an approach based on the period data type is that it allows existing databases that capture period information in a pair of datetime columns to take advantage of the SQL:2011 extensions more easily. Ever since DBMSs have been on the scene, users have been building their own solutions for handling temporal data as part of their application logic. Since most DBMSs do not support a period type, applications dealing with temporal data have tended to capture the period information using a pair of columns of datetime data type. It would be very expensive for users invested in such solutions to replace them with a solution that uses a single column of period type.

2. Interestingly, SQL:2011 manages to provide this support without actually defining or using the terms "temporal data" or "temporal table".

2.2 Application-time period tables

Application-time period tables are intended for meeting the requirements of applications that are interested in capturing time periods during which the data is believed to be valid in the real world. A typical example of such applications is an insurance application, where it is necessary to keep track of the specific policy details of a given customer that are in effect at any given point in time.

A primary requirement of such applications is that the user be put in charge of setting the start and end times of the validity period of rows, and the user be free to assign any time values, either in the past, current or in the future, for the start and end times. Another requirement of such applications is that the user be permitted to update the validity periods of the rows as errors are discovered or new information is made available.

Any table that contains a period definition with a user-defined name is an application-time period table. For example:

```
CREATE TABLE Emp(
ENo INTEGER,
EStart DATE,
EEnd DATE,
EDept INTEGER,
PERIOD FOR EPeriod (EStart, EEnd)
)
```

Users can pick any name they want for the name of the period as well as for the names of columns that act as the start and end columns of the period. The data types of the period start and end columns must be either DATE or a timestamp type, and data types of both columns must be the same.

The conventional INSERT statement provides sufficient support for setting the initial values of application-time period start and end columns. For example, the following INSERT statement inserts one row into the `Emp` table:

```
INSERT INTO Emp
VALUES (22217,
        DATE '2010-01-01',
        DATE '2011-11-12', 3)
```

The resulting table looks as shown below (assuming it was empty before):

Eno	EStart	EEnd	EDept
22217	2010-01-01	2011-11-12	3

The conventional UPDATE statement can be used to modify the rows of application-time period tables (including the application-time period start and end times). Similarly, the conventional DELETE statement can be used to delete rows of application-time period tables.

A new feature in SQL:2011 is the ability to specify changes that are effective within a specified period. This is provided by a syntactic extension to both UPDATE and DELETE statements that lets users specify the period of interest. For example, the following UPDATE statement changes the department of the employee whose number is 22217 to 4 for the period from Feb. 3, 2011 to Sept. 10, 2011:

```
UPDATE Emp
  FOR PORTION OF EPeriod
    FROM DATE '2011-02-03'
    TO DATE '2011-09-10'
SET EDept = 4
WHERE ENo = 22217
```

To execute this statement, the DBMS locates all rows whose application-time period overlaps the period *P* from Feb. 3, 2011 to Sept. 10, 2011. Recall that periods follow closed-open semantics in SQL:2011, so *P* includes Feb. 3, 2011 but excludes Sept. 10, 2011. Any overlapping row whose application-time period is contained in *P* is simply updated. If an overlapping row whose application-time period has a portion either strictly before or strictly after *P*, then that row gets split into two or three contiguous rows depending on the extent of overlap, and of these, the row whose application-time period is contained in *P* is updated. For example, suppose the following is the only overlapping row:

ENo	EStart	EEnd	EDept
22217	2010-01-01	2011-11-12	3

Note that the application-time period of the above row extends beyond *P* at both ends. The result of the UPDATE statement will be these three rows:

ENo	EStart	EEnd	EDept
22217	2010-01-01	2011-02-03	3
22217	2011-02-03	2011-09-10	4
22217	2011-09-10	2011-11-12	3

In this example, the row whose EDept value is updated to 4 is regarded as the original row and hence, UPDATE triggers fire for this row. The other two rows are regarded as newly inserted rows, so INSERT triggers fire for them.

The DELETE statement is similarly enhanced with FOR PORTION OF syntax to facilitate deletes that are only effective within a specified period. For example, the following DELETE statement removes the employee whose number is 22217 for the period from Feb. 3, 2011 to Sept. 10, 2011:

```
DELETE Emp
  FOR PORTION OF EPeriod
    FROM DATE '2011-02-03'
    TO DATE '2011-09-10'
WHERE ENo = 22217
```

Similar to the UPDATE example, any row whose application-time period is contained in *P* from Feb. 3, 2011 to Sept. 10, 2011 is simply deleted. If an overlapping row whose application-time period has a portion either strictly before or strictly after *P*, then that row gets split into two or three contiguous rows, and of these, the row whose application-time period is contained in *P* is deleted. For example, suppose the following is the only overlapping row:

ENo	EStart	EEnd	EDept
22217	2010-01-01	2011-11-12	3

The result of the statement will be these two rows:

ENo	EStart	EEnd	EDept
22217	2010-01-01	2011-02-03	3
22217	2011-09-10	2011-11-12	3

In this example, the result is the deletion of the original row and the insertion of two new rows; DELETE triggers fire for the deleted row and INSERT triggers fire for the newly inserted rows.

2.2.1 Primary keys on application-time period tables

The last section gave an example of an Emp table in which one might expect that ENo is the primary key. However, looking at the sample result of the UPDATE statement, there are three rows all with ENo 22217. This example shows that the primary key must also include the application-time period columns EStart and EEnd.

Simply adding `EStart` and `EEnd` to the primary key will not be sufficient though. Consider the following data:

ENo	EStart	EEnd	EDept
22217	2010-01-01	2011-09-10	3
22217	2010-02-03	2011-11-12	4

The triples (22217, 2010-01-01, 2011-09-10) and (22217, 2010-02-03, 2011-11-12) are not duplicates so they would be acceptable values for a conventional primary key on these three columns. But note that the application-time periods of these rows overlap. Semantically, this says that the employee with `ENo` 22217 belongs to two departments, 3 and 4, during the period from Feb. 3, 2010 through Sept. 10, 2011. Perhaps the user wishes to allow an employee to belong to two departments; however, the more typical requirement is that an employee belongs to exactly one department at any given time. To achieve that, it must be possible to forbid overlapping application-time periods, which can be specified with this syntax:

```
ALTER TABLE Emp
ADD PRIMARY KEY (ENo,
 EPeriod WITHOUT OVERLAPS)
```

With this primary key definition, the sample data is prohibited as a constraint violation.

2.2.2 Referential constraints on application-time period tables

Continuing the preceding example, suppose there is another table with the following definition:

```
CREATE TABLE Dept(
DNo INTEGER,
DStart DATE,
DEnd DATE,
DName VARCHAR(30),
PERIOD FOR DPeriod (DStart, DEnd),
PRIMARY KEY (DNo,
  DPeriod WITHOUT OVERLAPS)
)
```

Assume also that we want to make sure that at every point in time, every value in `EDept` column corresponds to some value of `DNo` column in `Dept` table, i.e., every employee at every point in time during her employment belongs to a department that actually exists at that point in time. How should this work? Let's look at some sample data. Assume the `Emp` table contains the following rows:

ENo	EStart	EEnd	EDept
22218	2010-01-01	2011-02-03	3
22218	2011-02-03	2011-11-12	4

Assume the `Dept` table contains the following rows:

DNo	DStart	DEnd	DName
3	2009-01-01	2011-12-31	Test
4	2011-06-01	2011-12-31	QA

Looking strictly at the values of `EDept` column of the `Emp` table and the `DNo` column of the `Dept` table, we may conclude that the conventional referential integrity constraint involving the two tables is satisfied. But note that the employee with `ENo` 22218 is assigned to the department with `DNo` 4 from Feb. 3, 2011 to Nov. 12, 2011, but there is no department with `DNo` 4 for the period from Feb. 3, 2011 to June 1, 2011. Clearly, this violates our requirement that every value of `EDept` column in `Emp` table corresponds to some value of `DNo` column in `Dept` table at every point in time. To disallow such a situation, it must be possible to forbid a row in a child table whose application-time period is not contained in the application-time period of a matching row in the parent table, which can be specified with this syntax:

```
ALTER TABLE Emp
ADD FOREIGN KEY
   (Edept, PERIOD EPeriod)
   REFERENCES Dept
   (DNo, PERIOD DPeriod)
```

With this referential constraint definition, the sample data is prohibited as a constraint violation.

More generally, for a given child row, it is not necessary that there exists exactly one matching row in the parent table whose application-time period contains the application-time period of the child row. As long as the application-time period of a row in the child table is contained in the union of application-time periods of two or more contiguous matching rows in the parent table, the referential constraint is considered satisfied.

2.2.3 Querying application-time period tables

In SQL:2011, application-time period tables can be queried using the regular query syntax. For example, to retrieve the department where the employee 22217 worked as of January 2, 2011, one can express the query as:

```
SELECT Name, Edept
FROM Emp
WHERE ENo = 22217
  AND EStart <= DATE '2011-01-02'
  AND EEnd > DATE '2011-01-02'
```

A simpler way to formulate the above query would be to employ one of the period predicates provided in SQL:2011 for expressing conditions involving periods: CONTAINS, OVERLAPS, EQUALS, PRECEDES, SUCCEEDS, IMMEDIATELY PRECEDES, and IMMEDIATELY SUCCEEDS. For example, the above query could also be expressed using the CONTAINS predicate, as shown below:

```
SELECT Ename, Edept
FROM Emp
WHERE ENo = 22217 AND
  EPeriod CONTAINS DATE '2011-01-02'
```

If one wanted to know all the departments where the employee whose number is 22217 worked during the period from January 1, 2010 to January 1, 2011, one could formulate the query as:

```
SELECT Ename, Edept
FROM Emp
WHERE ENo = 22217
  AND EStart < DATE '2011-01-01'
  AND EEnd > DATE '2010-01-01'
```

Note that the period specified in the above query uses the closed-open model, i.e., the period includes January 1, 2010 but excludes January 1, 2011. Alternatively, the same query could be expressed using the OVERLAPS predicate as:

```
SELECT Ename, Edept
FROM Emp
WHERE ENo = 22217 AND
  EPeriod OVERLAPS
      PERIOD (DATE '2010-01-01',
              DATE '2011-01-01')
```

Period predicates are functionally similar to (but not identical to) the well-known Allen's interval operators [11]. The correspondence between SQL's period predicates and Allen's operators is as follows:

- The predicate "X OVERLAPS Y" in SQL:2011 is equivalent to the Boolean expression using Allen's operators "(X overlaps Y) OR (X overlapped_by Y) OR (X during Y) OR (X contains Y) OR (X starts Y) OR (X started_by Y) OR (X finishes Y) OR (X finished_by Y) OR (X equal Y)". Note that Allen's overlaps operator is not a true test of period overlap. Intuitively, two periods are considered overlapping if they have at least one time point in common. This is not true for Allen's overlaps operator. In contrast, SQL:2011's OVERLAPS predicate is a true test of period overlap. Also, SQL:2011's OVERLAPS predicate is symmetric, i.e, if "X OVERLAPS Y" is true, then "Y OVERLAPS X" is also true. This is again not true for Allen's overlaps operator.
- The predicate "X CONTAINS Y" in SQL:2011 is equivalent to the Boolean expression using Allen's operators "(X contains Y) OR (X starts Y) OR (X finishes Y) OR (X equal Y)". Note that Allen's contains operator is not a true test of period containment. Intuitively, period X is considered containing period Y if every time point in Y is also in X. This is not true for Allen's contains operator. In contrast, SQL:2011's CONTAINS predicate is a true test of period containment.
- The predicate "X PRECEDES Y" in SQL:2011 is equivalent to the Boolean expression using Allen's operators "(X before Y) OR (X meets Y)".
- The predicate "X SUCCEEDS Y" in SQL:2011 is equivalent to the Boolean expression using Allen's operators "(X after Y) OR (X met_by Y)".
- The predicates "X EQUALS Y", "X IMMEDIATELY PRECEDES Y", and "X IMMEDIATELY SUCCEEDS Y" in SQL:2011 are equivalent to the Allen's operators "X equal Y", "X meets Y", and "X met_by Y", respectively.

2.3 System-versioned tables

System-versioned tables are intended for meeting the requirements of applications that must maintain an accurate history of data changes either for business reasons, legal reasons, or both. A typical example of such applications is a banking application, where it is necessary to keep previous states of customer account information so that customers can be provided with a detailed history of their accounts. There are also plenty of examples where certain institutions are required by law to preserve historical data for a specified length of time to meet regulatory and compliance requirements.

A key requirement of such applications is that any update or delete of a row must automatically preserve the old state of the row before performing the update or delete. Another important requirement is that the system, rather than the user, maintains the start and end times of the periods of the rows, and that users be unable to modify the content of historical rows or the periods associated with any of the rows. Any updates to the periods of rows in a system-versioned table must be performed only by the system as a result of updates to the non-period columns of the table or as a result of row deletions. This provides the guarantee that the recorded history of data changes cannot be tampered with, which is critical to meet auditing and compliance regulations.

Any table that contains a period definition with the standard-specified name, `SYSTEM_TIME`, and includes the keywords `WITH SYSTEM VERSIONING` in its definition is a system-versioned table. Similar to application-time period tables, users can pick any name they want for the names of columns that act as the start and end columns of the `SYSTEM_TIME` period. Though SQL:2011 allows the data types of the period start and end columns to be either DATE or a timestamp type (as long as the data types of both columns are the same), in practice, most implementations will provide the TIMESTAMP type with the highest fractional seconds precision as the data type for the system-time period start and end columns. For example:

```
CREATE TABLE Emp
 ENo INTEGER,
 Sys_start TIMESTAMP(12) GENERATED
  ALWAYS AS ROW START,
 Sys_end TIMESTAMP(12) GENERATED
  ALWAYS AS ROW END,
 EName VARCHAR(30),
 PERIOD FOR SYSTEM_TIME (Sys_start,
  Sys_end)
) WITH SYSTEM VERSIONING
```

Similar to application-time periods, system-time periods use closed-open period model. At any given point in time, a row in a system-versioned table is regarded as *current system row* if the system-time period of that row contains the current time. A row that is not a current system row is regarded as a *historical system row.*

System-versioned tables differ from application-time period tables in the following respects:

1) In contrast to the application-time period tables, users are not allowed to assign or change the values of `Sys_start` or `Sys_end` columns; they are assigned (and changed) automatically by the database system. This is the reason why the definitions of `Sys_start` or `Sys_end` columns must include the keywords `GENERATED ALWAYS.`
2) `INSERT` into a system-versioned table automatically sets the value of `Sys_start` column to the *transaction timestamp*, a special value associated with every transaction[3], and sets the value of `Sys_end` column to the highest value of the column's data type. For example, assume that the following `INSERT` statement executed in a transaction whose transaction timestamp is 2012-01-01 09:00:00[4]:

```
INSERT INTO Emp (ENo, EName)
  VALUES (22217, 'Joe')
```

The resulting table looks as shown below (assuming it was empty before):

ENo	Sys_Start	Sys_End	EName
22217	2012-01-01 09:00:00	9999-12-31 23:59:59	Joe

3) `UPDATE` and `DELETE` on system-versioned tables only operate on current system rows. Users are not allowed to update or delete historical system rows. Users are also not allowed to modify the system-time period start or the end time of both current system rows and historical system rows.
4) `UPDATE` and `DELETE` on system-versioned tables result in the automatic insertion of a historical system row for every current system row that is updated or deleted.

An `UPDATE` statement on a system-versioned table first inserts a copy of the old row with its system-time period end time set to the transaction timestamp, indicating that the row ceased to be current as of the transaction timestamp. It then updates the row while changing its system-period start time to the transaction timestamp, indicating that the updated row to be the current system row as of the transaction timestamp. For example, suppose the current system row with `ENo` 22217 is as shown below:

ENo	Sys_Start	Sys_End	EName
22217	2012-01-01 09:00:00	9999-12-31 23:59:59	Joe

The following `UPDATE` statement changes the name of the employee whose number is 22217 from Joe to Tom effective from the transaction timestamp of the transaction in which the `UPDATE` statement was executed:

```
UPDATE Emp
SET EName = 'Tom'
WHERE ENo = 22217
```

A historical system row that corresponds to the state of the row prior to the update is first inserted and then

3. SQL:2011 leaves it up to SQL-implementations to pick an appropriate value for the transaction timestamp of a transaction, but it does require the transaction timestamp of a transaction to remain fixed during the entire transaction.

4. Note that we are not showing the fractional part of seconds in any of the examples in this Section.

the update is performed. Assuming the above statement is executed in a transaction with the transaction timestamp 2012-02-03 10:00:00, the final result will be these two rows:

ENo	Sys_Start	Sys_End	EName
22217	2012-01-01 09:00:00	2012-02-03 10:00:00	Joe
22217	2012-02-03 10:00:00	9999-12-31 23:59:59	Tom

In this example, the row whose name is Tom is the updated row; `UPDATE` triggers fire for this row. Note that the insertion of historical system rows does not fire any `INSERT` triggers for the inserted rows. Note also that historical system rows created as a result of sequence of updates for a given row form one contiguous chain without any gap between their system-time periods.

A `DELETE` statement on a system-versioned table does not actually delete the qualifying rows; instead it changes the system-time period end time of those row to the transaction timestamp, indicating that those rows ceased to be current as of the transaction timestamp. For example, suppose that the current system row with ENo 22217 is as shown below:

ENo	Sys_Start	Sys_End	EName
22217	2012-01-01 09:00:00	9999-12-31 23:59:59	Joe

The following `DELETE` statement simply changes the system-time period end time of the current system row for the employee 22217 to the transaction timestamp of the transaction in which the `DELETE` statement was executed:

```
DELETE FROM Emp
WHERE ENo = 22217
```

Assuming the above statement is executed in a transaction with the transaction timestamp 2012-06-01 00:00:00, the final result will be the following row:

ENo	EStart	EEnd	EName
22217	2012-01-01 09:00:00	2012-06-01 00:00:00	Joe

In this example, `DELETE` triggers fire for the row selected for deletion.

Note that in contrast to the application-time period tables, `FOR PORTION OF SYSTEM_TIME` is not needed (and hence not allowed) for the `UPDATE` and `DELETE` statements on system-versioned tables.

2.3.1 Primary key and referential constraints on system-versioned tables

The definition and enforcement of constraints on system-versioned tables is considerably simpler than the definition and enforcement of constraints on application-time period tables. This is because constraints on system-versioned tables need only be enforced on the current system rows. Historical system rows in a system-versioned table form immutable snapshots of the past. Any constraints that were in effect when a historical system row was created would have already been checked when that row was a current system row, so there is never any need to enforce constraints on historical system rows. Consequently, there is no need to include the system-period start and end columns or the period name in the definition of primary key and referential constraints on system-versioned tables. For example, the following ALTER TABLE statement specifies `ENo` column as the primary key of `Emp` table:

```
ALTER TABLE Emp
ADD PRIMARY KEY (ENo)
```

The above constraint ensures there exists exactly one current system row with a given `ENo` value.

Similarly, the following ALTER TABLE statement specifies a referential constraint between `Emp` and `Dept` tables:

```
ALTER TABLE Emp
ADD FOREIGN KEY (Edept)
   REFERENCES Dept (DNo)
```

The above constraint is again enforced only on the current system rows of `Emp` and `Dept` tables.

2.3.2 Querying system-versioned tables

Because system-versioned tables are intended primarily for tracking historical data changes, queries on system-versioned tables often tend to be concerned with retrieving the table content as of a given point in time or between any two given points in time. SQL:2011 provides three syntactic extensions for this specific purpose. These are allowed only in queries on system-versioned tables.

The first extension is the `FOR SYSTEM_TIME AS OF` syntax that is useful for querying the table content as of a specified point in time. For example, the following query retrieves the rows of `Emp` that were current as of Jan. 2, 2011:

```
SELECT ENo,EName,Sys_Start,Sys_End
FROM Emp FOR SYSTEM_TIME AS OF
  TIMESTAMP '2011-01-02 00:00:00'
```

The above query returns all rows whose system-time period start time is less than or equal to the specified timestamp and whose system-time period end time is greater than the specified timestamp.

The second and third extensions allow for retrieving the content of a system-versioned table between any two points in time. The following query returns all rows that were current starting from `TIMESTAMP '2011-01-02 00:00:00'` up to (but not including) `TIMESTAMP '2011-12-31 00:00:00'`:

```
SELECT ENo,EName,Sys_Start,Sys_End
FROM Emp FOR SYSTEM_TIME FROM
 TIMESTAMP '2011-01-02 00:00:00'TO
 TIMESTAMP '2011-12-31 00:00:00'
```

In contrast, the following query returns all rows that were current starting from `TIMESTAMP '2011-01-02 00:00:00'` up to (and including) `TIMESTAMP '2011-12-31 00:00:00'`:

```
SELECT ENo,EName,Sys_Start,Sys_End
FROM Emp FOR SYSTEM_TIME BETWEEN
 TIMESTAMP '2011-01-02 00:00:00'AND
 TIMESTAMP '2011-12-31 00:00:00'
```

Note that the period specified in the (`FROM` ... `TO` ...) corresponds to a closed-open period model while the period specified in the (`BETWEEN` ... `AND` ...) corresponds to a closed-closed period model.

If a query on system-versioned tables does not specify any of the above three syntactic options, then that query is assumed to specify `FOR SYSTEM_TIME AS OF CURRENT_TIMESTAMP` by default and the query returns only the current system rows as the result. For example, the following query returns only the current system rows of `Emp` table:

```
SELECT ENo,EName,Sys_Start,Sys_End
FROM Emp
```

The choice of returning current systems rows as the default is especially suited for those applications where retrieval of current system rows is the most frequent operation. In addition, it also helps with the database migration in that applications running on non-system-versioned tables would continue to work and produce the same results when those tables are converted to system-versioned tables.

Finally, to retrieve both current and historical system rows of a system-versioned table, one can use a query of the kind shown below:

```
SELECT ENo,EName,Sys_Start,Sys_End
FROM Emp FOR SYSTEM_TIME FROM
 TIMESTAMP '0001-01-01 00:00:00' TO
 TIMESTAMP '9999-12-31 23:59:59'
```

2.4 Bitemporal tables

A table may be both a system-versioned table and an application-time period table[5]. For example:

```
CREATE TABLE Emp(
  ENo INTEGER,
  EStart DATE,
  EEnd DATE,
  EDept INTEGER,
  PERIOD FOR EPeriod (EStart, EEnd),
  Sys_start TIMESTAMP(12) GENERATED
      ALWAYS AS ROW START,
  Sys_end TIMESTAMP(12) GENERATED
      ALWAYS AS ROW END,
  EName VARCHAR(30),
  PERIOD FOR SYSTEM_TIME
      (Sys_start, Sys_end),
  PRIMARY KEY (ENo,
      EPeriod WITHOUT OVERLAPS),
  FOREIGN KEY
      (Edept, PERIOD EPeriod)
   REFERENCES Dept
      (DNo, PERIOD DPeriod)
) WITH SYSTEM VERSIONING
```

Rows in such tables are associated with both the system-time period and the application-time period. Such tables are very useful for capturing both the periods during which facts were believed to be true in the real world as well as the periods during which those facts were recorded in the database. For example, while employed, an employee may change names. Typically the name changes legally at a specific time (for example, a marriage) but the name is not changed in the database concurrently with the legal change. In that case, the system-time period automatically records when a particular name is known to the database, and the application-time period records when the name was legally effective. Successive updates to bitemporal tables can journal complex twists and turns in the state of knowledge captured by the database.

Bitemporal tables combine the capabilities of both system-versioned and application-time period tables. As in the case of application-time period tables, the user is in charge of supplying values for the application-time period start and end columns. As in the case of system-versioned tables, `INSERT` into such a table automatically sets the value of system-time period start column to the transaction timestamp, and the value of system-

5. Though SQL:2011 does not define any specific term for such tables, we use the term "bitemporal tables" in keeping with its use in the literature as well as in some products.

time period end column to the highest value of the column's data type.

As in the case of application-time period tables, both the conventional `UPDATE` statement as well as `UPDATE` with `FOR PORTION OF` *app-period*, where *app-period* is the name of application-time period, can be used to modify the rows of bitemporal tables. Similarly, the conventional `DELETE` statement as well as `DELETE` with `FOR PORTION OF` *app-period* can be used to delete rows from bitemporal tables. As in the case of system-versioned tables, only current rows in system-time can be updated or deleted and a historical system row is automatically inserted for every current system row that is updated or deleted.

Queries on bitemporal tables can specify predicates on both application-time periods as well as system-time periods to qualify rows that will be returned as the query result. For example, the following query returns the department where the employee 22217 worked as of December 1, 2010, recorded in the database as of July 1, 2011:

```
SELECT ENo, EDept
FROM Emp FOR SYSTEM_TIME AS OF
    TIMESTAMP '2011-07-01 00:00:00'
WHERE ENo = 22217 AND
 EPeriod CONTAINS DATE '2010-12-01'
```

2.5 Future directions

Though SQL:2011 has incorporated several significant extensions for managing temporal data, there is certainly room for additional extensions. These are left as Language Opportunities for future versions of the standard. Here is a partial list of such extensions:

- Support for period joins, i.e., joining a row from one table with a row from another table such that their application-time or system-time periods satisfy a condition such as overlap. Note that it is possible to do an inner join of this kind using SQL:2011's `OVERLAPS` predicate, but outer joins require support for additional syntax built into the language.
- Support for period aggregates and period grouped queries that take into account application-time or system-time periods of rows.
- Support for period `UNION`, `INTERSECT` and `EXCEPT` operators that take into account application-time or system-time periods of rows.
- Support for period normalization that produces semantically-equivalent minimal set of rows for a given table by combining contiguous rows that have exactly the same values in non-period columns.
- Support for multiple application-time periods per table.
- Support for non-temporal periods.

3. Comparison with previous temporal proposals

Earlier, we alluded to the fact that the SQL committee had initiated a temporal project that was eventually cancelled around 2001. We list below some of the differences between the approach taken by the previous proposals and the approach taken by SQL:2011 extensions:

- In previous proposals, the period information was associated with the rows of temporal tables using an unnamed hidden column. This design was motivated by the notion of *temporal upward compatibility* [12], which required a temporal table and its equivalent non-temporal table to have exactly the same number of columns. One major drawback of this approach is that it is incompatible with SQL's notion of tables, which requires all information associated with the rows of a table to be captured explicitly as (*and only as*) column values. The other drawback was that queries of the form "SELECT * FROM T", where T is a temporal table, did not return the period information associated with the rows of T in the query result. If users wanted to access the period information associated with the rows, they were forced to include invocations of special built-in functions in the select list of a query for that purpose. These built-in functions operated on the range variables associated with temporal tables in a query expression, and returned the period value associated with the rows pointed to by those range variables. In contrast, the period information is associated with the rows of temporal tables using explicit, user-defined columns in SQL:2011. Also, the period information associated with the rows of a temporal table can be accessed in SQL:2011 simply by including the corresponding period start and end columns in the select list of a query.
- The previous proposals resorted to a controversial technique of prefixing queries, constraints, and insert/update/delete statements with the so-called *statement modifiers* for changing their normal semantics [12]. Unfortunately, previous proposals contained no clear rules specifying the semantics of constructs prefixed with these statement modifiers, so it was hard to figure out the end result [9]. In contrast, SQL:2011 provides a small set of syntactic extensions with clearly-specified scope and semantics.
- In previous proposals, query expressions, constraint definitions, and insert/update/delete statements expressed without the statement modifier prefixes were assumed to operate only on the current rows. This applied to both transaction time tables and valid time tables. While this made sense for transaction time tables, it did not make much sense for valid time tables. For instance, users were allowed to insert into valid time

tables only those rows whose valid time period started with the current time. In fact, there was no way for users to insert rows into valid time tables whose validity periods were either in the past or in the future. In contrast, query expressions, constraint definitions, and insert/update/delete statements on application-time period tables in SQL:2011 operate on the entire table content and follow the standard semantics. Also, SQL:2011 allows users to specify any time values they desire for the application-time period start and end columns as part of the INSERT statement on application-time period tables.

• The previous proposals relied on adding special syntax to the table definition for creating temporal tables (AS TRANSACTION TIME for transaction-time support and AS VALIDTIME for valid-time support). Consequently, supporting additional periods in previous approach would have required extending the table definition syntax every time a new period was added. In contrast, supporting additional periods requires no new syntax in SQL:2011.

4. Acknowledgements

The authors thank Fred Zemke and Matthias Nicola for their valuable comments on the prior versions of this article.

5. References

[1] Fred Zemke, "What's new in SQL:2011", SIGMOD Record, Vol. 41, No. 1, March 2012, pp. 67-73, http://www.sigmod.org/publications/sigmod-record/1203/pdfs/10.industry.zemke.pdf/

[2] Yu Wu, Sushil Jajodia, X. Sean Wang, "Temporal Database Bibliography Update", In Temporal Databases: Research and Practice, O. Etzion, S. Jajodia, and S.Sripada, eds., Springer, 1998

[3] Richard Snodgrass, "Developing Time-Oriented Database Applications in SQL", Morgan Kaufmann, 1999

[4] C. J. Date, Hugh Darwen, Nikos A. Lorentzos, "Temporal Data and the Relational Model", Morgan Kaufman, 2003

[5] Cynthia Saracco, Matthias Nicola, Lenisha Gandhi, "A matter of time: Temporal data management in DB2 10", April 2012, http://www.ibm.com/developerworks/data/library/techarticle/dm-1204db2temporaldata/

[6] Kevin Jerrigan, "Oracle Total Recall with Oracle Database 11g Release 2", September 2009, http://www.oracle.com/us/products/database/security/total-recall-whitepaper-171749.pdf

[7] Gregory Sannik, Fred Daniels, "Enabling the Temporal Data Warehouse", September 2010, http://www.teradata.com/white-papers/

[8] Richard Snodgrass (Ed.), "The TSQL2 Temporal Query Language", Kluwer Academic Publishers, 1995

[9] Hugh Darwen, C.J. Date, "An overview and Analysis of Proposals Based on the TSQL2 Approach", In Date on Database: Writings 2000-2006, C.J. Date, Apress, 2006, also avaliable in http://www.dcs.warwick.ac.uk/~hugh/TTM/OnTSQL2.pdf

[10] ISO/IEC 9075-2:2011, *Information technology—Database languages—SQL—Part 2: Foundation (SQL/Foundation)*, 2011

[11] James F. Allen, "Maintaining knowledge about temporal intervals", Communications of ACM, Vol. 26, No. 11, November 1983

[12] Michael Bohlen, Christian Jensen, Richard Snodgrass, "Temporal Statement Modifiers", ACM Trans. on Database Systems, Vol. 25, No. 4, December 2000