A Comparison of Different Forms of Temporal Data Management

Florian Künzner and Dušan Petković(✉)

University of Applied Sciences
Rosenheim, Hochschulstr. 1, 83024, Germany
flo.kuenzner@gmx.net, petkovic@fh-rosenheim.de
http://www.fh-rosenheim.de

Abstract. Recently, the ANSI committee for the standardization of the SQL language has published the specification for temporal data support. This new ability allows users to create and manipulate temporal data in a significantly simpler way instead of implementing the same features using triggers and database applications. In this article we examine the creation and manipulation of temporal data using built-in temporal logic and compare its performance with the performance of equivalent hand-coded applications. For this study, we use an existing commercial database system, which supports the standardized temporal data model.

Keywords: Temporal data · SQL:2011 · Performance · Trigger

1 Introduction

A database represents a model of objects of the real world. During the lifetime of an object stored in a database, its properties can change. For such objects it is necessary to consider their time-variant aspects. For this reason, it is an important task of database systems to support management of temporal data, i.e. that current, past and future values of time-variant attributes can be persistently stored.

Timestamps can be represented using time points, time intervals and set of time intervals. Time points are an infinite but countable ordered set, which is used to specify a time domain [16]. Time intervals specify a time domain of an entity as the continuous set of time instants. Interval-based temporal data models are introduced in TSQL2 [14] and in the SQL:2011 standard [6]. Sets of time intervals represent time domain as a finite union of time intervals [15]. Almost all existing temporal data models today are based upon time intervals.

Also, there are three fundamental and orthogonal kinds of time: user-defined, valid and transaction time. User-defined time is a time representation designed to meet the specific needs of users. Valid time specifies when certain conditions in the real world are, were or will be valid. Several models supporting valid time are described in [15]. Transaction time automatically captures changes made to the state of time-variant data in a database. This time dimension represents the time period during which an instance is recorded in the database. Taxonomy for classifying databases in terms of valid time and transaction time has been developed in [13].

The first proposal for standardization of temporal data for database systems has been submitted in 1995 and was based on [14]. This proposal showed some shortcomings and was criticized in an article [3], thus failed to get a support of the SQL standardization committee. Another proposal [9] has been submitted, but the committee did not accept it. The next attempt to standardize temporal data as a part of the SQL specification started in 2006 and ended successfully with the publishing of the SQL:2011 standard [7]. The standard has adopted a model, where a time period represents all time granules starting from and including the start time, and ending with the last time granule before the end time.

The standard introduces three different forms of tables [11]: application-time period tables, system-versioned tables and bitemporal tables. An application-time period table captures time periods during which data are valid in the real world. For this reason, the user is responsible for setting the start and end times of each time-variant attribute. Also, the user modifies the validity periods of rows, when an error is detected. Generally, the valid time support is provided by these tables.

One of main requirements for system-versioning table is that any modification of rows immediately preserves the old state of these rows before executing the UPDATE or DELETE statement. By specifying a table with system-versioning, the user tells the system to immediately capture changes made to the state of tables rows and, at the same time to memorize the old state of the same rows. Therefore, the time period of a row in a system-versioning table begins at the time in which the row was inserted into the table and ends when the row was deleted or updated.

Bitemporal tables store and manage data with valid as well as with transaction time. Therefore, a bitemporal table is a union of data from a corresponding application-time period table and system-versioned table, and rows in such tables are associated with both application-time period as well as with system-time period. (Note that SQL:2011 does not use any particular name for these tables. In this article, the phrase bitemporal is used in accordance to its use in the literature.)

The aim of this paper is to examine whether the built-in support for temporal logic reduces execution time in relation to hand-coded database applications. Our tests showed that in the most cases execution times of built-in code is ca. 1.5 times quicker than execution time of the corresponding triggers and stored procedures.

For this study, we use an existing commercial database system, IBM DB2, which supports the standardized temporal logic [12]. Besides DB2, Oracle Version 12c as well as Teradata support temporal logic. Teradatas temporal support does not correspond to the specification in the SQL:2011 standard, while Oracles implementation is incomplete in relation to the DB2 support of this specification.

1.1 Related Work

Temporal databases have been a research topic since more than 25 years. During this time, numerous temporal models and query languages have been proposed. An annotated bibliography on temporal aspects can be found in [5]. The glossary of temporal database concepts is given in [8]. Taxonomy for the classification of temporal databases according to time dimensions has been developed in [13]. According to this taxonomy, Gadias work, described in [15] is a temporal data model concerning valid time. Ben-Zvi proposed the first data model for bitemporal databases, a temporal query language, storage architecture, indexing, recovery, concurrency, synchronization, and its implementation [4]. Snodgrass attached four implicit attributes to each time-varying relation, and presented a corresponding temporal query language [15]. The bitemporal data model, BCDM [1] forms the basis for the Temporal Structured Query Language. The article [17] discusses the standardization of database systems from another angle of view.

Performance issues in relation to temporal data have been investigated in several papers. In her work [1], Attay compared attribute and tuple timestamping in relation to the BCDM data model. Our work is similar to the work at IBM described in [12]. In contrast to our tests, the authors in [12] compare implementation of a subset of valid time capabilities with built-in temporal data management versus hand-coded implementation of equivalent logic. The built-in implementation of temporal logic reduced coding requirements by ca. 90% in relation to the implementation of the same in corresponding applications. In relation to lines of code, triggers and Java applications required 16 and 45 times more LOCs, respectively.

1.2 Roadmap

The rest of this article is organized in the following way: Section 2 deals with issues in relation to application-time period tables. Different cases concerning integrity constraints as well as DML statements on such tables are analyzed and compared. Section 3 discusses two different cases in relation to DML statements of system-versioned tables. In Section 4 we show cases in relation to integrity constraints and DML statements concerning bitemporal tables. Section 5 summarizes all experimental results from Chapters 2-4. The last section gives conclusions and discusses future work. The article has an appendix (Supplement), which contains the code of all triggers and stored procedures used for testing.

Each of the following three sections (Sections 2-4) has the same structure. First, we give basic specifications of all tables used in the following subsections. After that, the issues in relation to key integrity are discussed. In the final subsection, the discussion of DML statements is given.

2 Application-Time Period Tables

The creation of application-time period tables requires that the user explicitly defines two time-variant columns, which specify the start and end times of the period of each row. Both columns must be defined with the NOT NULL property and their type can be any date type (DATE or TIMESTAMP). The PERIOD clause, which is a new extension to SQL, instructs the system to use these time-variant columns to track the start and end points of time values for each row. (This clause implicitly enforces the constraint that start_time < end_time.)

2.1 Basic Specifications

The **employees** table is an SQL table used for testing of a logic implemented in hand-coded applications. The corresponding CREATE TABLE statement is given in Example 1. Similarly, the **T_employees** table is the corresponding table with the built-in temporal logic and handles valid time. (IBM calls such a table "table with business time".) The corresponding statement is given in Example 2.

Example 1

```
CREATE TABLE employees (
    E_Id INT NOT NULL, E_Name CHAR(20) NOT NULL,
    E_Start DATE NOT NULL, E_End DATE NOT NULL,
    E_Dept INT,
    PRIMARY KEY (E_Id, E_Start));
-- Start_Time <= End_Time ->  must be explicitly checked
ALTER TABLE employees ADD CONSTRAINT emp_check
    CHECK(E_Start <= E_End);
```

Example 2

```
CREATE TABLE T_employees (
    E_Id INT NOT NULL, E_Name CHAR(20) NOT NULL,
    E_Start DATE NOT NULL, E_End DATE NOT NULL,
    E_Dept INT,
    PERIOD BUSINESS_TIME (E_Start, E_End),
    PRIMARY KEY (E_Id, BUSINESS_TIME WITHOUT OVERLAPS));
```

For our tests, each of these tables is consecutively loaded with 100, 1000, 10.000, and 100.000 rows. Examples for loading data in both tables are given in Examples 3 and 4, respectively. As can be seen from these examples, both INSERT statement are equivalent. The reason is that in both CREATE TABLE statements the user has to specify start and end time of the corresponding time period.

Example 3

```
INSERT INTO employees
    VALUES (115, 'Dante', DATE '2012-01-01', DATE '2012-12-01', 10);
```

Example 4

```
INSERT INTO T_employees
    VALUES (115, 'Dante', DATE '2012-01-01', DATE '2012-12-01', 10);
```

2.2 Key Integrity Constraints on Application-Time Period Tables

Primary Key Integrity. There are two key integrity constraints: one concerning primary key (PK) and the other concerning foreign key i.e. referential integrity (RI). The first consequence of adding time support to an application-time period table is that the "convenient" primary key for the relational model is not sufficient for temporal logic and tables with time-variant columns have to consider the timestamp. In other words, the **E_Id** column in Example 2 does not suffice to guarantee uniqueness of rows in the **T_employees** table and the primary key specification has to include the **E_start** and **E_End** columns. However, this is still not enough for the uniqueness of rows, while we wish to specify that there can be only one **E_Id** value at any given time. In other words, we want that an employee belongs to exactly one department at the same time. For this reason, the standard specifies the WITHOUT OVERLAPS clause, which forbids overlapping of time periods. The first test we applied concerns the problem of primary key integrity. Example 5 tries to insert a row, which violates the constraint specified with the WITHOUT OVERLAPS clause in the CREATE TABLE statement of Example 2 and the already existing row inserted in Example 3.

Example 5

```
INSERT INTO T_employees
    VALUES (115, 'Dante', DATE '2012-01-01', DATE '2012-03-01', 10);
```

The same constraint for the convenient SQL table (**employees**) is shown in Example 6. The trigger in this example rejects the execution of INSERT statements that violate the PK constraint on the **employees** table. Note that this trigger is the only one, which is presented in the text of this article to demonstrate implementation of hand-coded applications. All other applications that will be discussed below can be found in the Supplement.

Example 6

```
-- the '!' sign is used as a delimiter
CREATE OR REPLACE TRIGGER EMPLOYEES_TRIGGER_INSERT
    BEFORE INSERT ON employees REFERENCING NEW AS newRow
    FOR EACH ROW
    BEGIN
        DECLARE i INTEGER;
        SELECT COUNT(*) INTO i
            FROM employees
            WHERE newRow.E_Id = E_Id
                AND newRow.E_End > E_Start
                AND newRow.E_Start < E_End;

         IF i > 0 THEN CALL
             RAISE_APPLICATION_ERROR(-20001, 'PK Times overlaps');
         END IF;
    END!
```

Referential Integrity. If we suppose that there is the **T_Departments** table, which is a parent table of the **T_employees** table, the convenient referential integrity involving these two tables cannot be satisfied, if the similar requirements from the last subsection are valid, i.e. that every value of the **E_Dept** column of the **T_employees** table corresponds to the department number column of the **T_Departments** table at every point in time. The SQL:2011 standard specifies the corresponding FOREIGN KEY clause (with the REFERENCES option), but the implementation of such a clause does not yet exist in IBM DB2 [10]. For this reason, a corresponding case cannot be tested at this moment.

2.3 DML Statements and Application-Time Period Tables

The semantics for DML statements differ significantly for application-time period tables in relation to the conventional SQL. Table 1 shows the differences for all three of these statements.

Table 1. The semantics of DML statements for application-time tables

	Conventional SQL	SQL Temporal
INSERT (one row)	Inserting the new row	Inserting a new row or prolonging the valid time of an existing row.
UPDATE (one row)	Updating the row	Shortening the valid time of the row and inserting a new row or prolonging valid time of an existing row.
DELETE (one row)	Deleting the row	Shortening the valid time of the row or deleting the row.

The case we examine concerns the DELETE statement. As can be seen from Table 1, the deletion of a row with a time-variant attribute means that its valid time is shortened or deleted. Example 7 shows such a DELETE statement for the **T_employees** table. After execution of this statement, the result contains two rows with **E_Id** = 115, one with the time period ending on 15.7.2012, and the other beginning on 15.8.2012.

Example 7

```
DELETE FROM T_employees FOR PORTION OF BUSINESS_TIME
    FROM DATE '2012-07-15' TO DATE '2012-08-15' WHERE E_Id = 115;
```

The example above uses the new clause - FOR PORTION -, which is the temporal extension of the DELETE (and UPDATE) statement. The Supplement shows the implementation of the corresponding stored procedure. (In this case, we used a stored procedure to implement temporal logic, because that way the explicit values of start and end times can be passed to the system using two parameters.)

3 System-Versioned Tables

System-versioned tables comprise system times, which are always updated, when the data modification statements are executed. These tables contain two additional columns, for system start time and system end time. In contrast to application-time period tables, the system maintains the start and end times of the periods of rows.

3.1 Basic Specifications

The **departments** table, created in Example 8 is the parent table of the **employees** table from Example 1. This table and the corresponding history table are used for testing a logic implemented in hand-coded applications and for transaction time. Similarly, the **S_departments** table is the corresponding system-versioned table with the built-in temporal logic. (IBM calls such a table ”table with system time”.) The corresponding CREATE TABLE statement is given in Example 9.

The two time-variant columns of the **S_departments** table (**D_Start** and **D_End**) are specified by the user, but their values are provided by the system. The third column, **TRANS_Start**, is used to track when the transaction first executed a statement that changes the tables data. The PERIOD clause has the same meaning as the clause with the same name for application-time period tables.

As can be seen from Example 9, IBM DB2 uses two tables, one for current data, (**S_departments**) and one for historical data (**Dept_history**). This is in contrast to the SQL:2011 specification, which defines only one table for current and historical data. Both DB2 tables are compatible to each other, and the system automatically moves the non-current data in the history table. Additionally, the ALTER TABLE statement in Example 9 alters the current table to enable versioning and identify the corresponding history table.

Example 8

```
CREATE TABLE departments(
    D_id INT NOT NULL, D_Name CHAR(20),
    D_Start TIMESTAMP(12), D_End TIMESTAMP(12),
    PRIMARY KEY (D_Id));
CREATE TABLE departments_History(
    D_id INT NOT NULL, D_Name CHAR(20),
    D_Start TIMESTAMP(12) NOT NULL, D_End TIMESTAMP(12) NOT NULL,
    PRIMARY KEY (D_Id, D_Start));
```

Example 9

```
CREATE TABLE S_departments(
    D_id INT NOT NULL, D_Name CHAR(20),
    D_Start TIMESTAMP(12) GENERATED ALWAYS AS ROW BEGIN NOT NULL,
    D_End TIMESTAMP(12) GENERATED ALWAYS AS ROW END NOT NULL,
    TRANS_Start GENERATED ALWAYS AS TRANSACTION
                START ID IMPLICITLY HIDDEN,
    PERIOD SYSTEM_TIME (D_Start, D_End), PRIMARY KEY (D_id));
CREATE TABLE Dept_History LIKE S_departments;
ALTER TABLE S_departments ADD VERSIONING USE HISTORY TABLE Dept_History;
```

The following examples show how rows can be inserted in both tables created in Example 8 and Example 9.

Example 10

```
INSERT INTO departments
    VALUES (1, 'D1', CURRENT TIMESTAMP, DATE '9999-12-31');
```

Example 11

```
INSERT INTO S_departments (D_Id, D_Name) VALUES ('1', 'D1');
```

The INSERT statements in Examples 10 and 11 are different, because in temporal logic, values for transaction start and end times are assigned by the system.

3.2 Key Integrity Constraints on System-Versioned Tables

The specification of key integrity constraints on system-versioned tables is significantly simpler than the specification of the same integrity constraints on application-time period tables. The reason is that these constraints (primary key constraint and referential integrity) must be enforced only on the current rows, because the same constraints have been already checked at the time where historical rows were current ones. For this reason, there is no need to include time-variant columns (begin system time and end system time) in the primary key and referential integrity definitions. In other words, the key integrity constraints on system-versioned tables are analogous to the same constraints in the conventional SQL.

3.3 DML Statements and System-Versioned Tables

The semantics for the INSERT, UPDATE and DELETE statements differ for system-versioned tables in relation to the conventional SQL. Table 2 shows the differences for all three of these statements.

Table 2. The semantics of DML statements in system-versioned tables

	Conventional SQL	SQL Temporal
INSERT	Inserting a new row	Inserting a new row
UPDATE (one row)	Modifying the row	Ending the currentness of that row and inserting a new row as a substitution for the ex-current row.
DELETE (one row)	Deleting the row	Ending the currentness of the row.

For system-versioned tables, our tests for DML concern the UPDATE (Example 12) and DELETE (Example 13) statements.

Example 12

```
UPDATE S_departments SET D_Name = 'D3' WHERE D_id = 1;
```

As can be seen from Table 2, the UPDATE statement in Example 12 will be replaced by an UPDATE and an INSERT statement: the former concerning the table with current values and the latter concerning the history table. The existing row in the **S_departments** table will be modified, with the new value for the **D_Name** column, and the start system time of the row will be set to the time, when the UPDATE statement has been performed. The historical row will have the end transaction time set to the time, when the UPDATE statement has been performed. (All other values of this row will be unchanged.)

Example 13

```
DELETE FROM S_departments WHERE D_id = 1;
```

The DELETE statement in Example 13 ends the currentness of the row with **D_id**=1. This means that two statements will be executed: a DELETE and an INSERT. In other words, the row will be deleted from the **S_departments** table and the new row will be inserted in the history table with the value of end transaction time set to the current time.

4 Bitemporal Tables

As already stated, valid time and transaction time represent two different kinds of time, which are orthogonal to each other. In the case that an application needs both of these dimensions, bitemporal tables are used. For this reason, the main requirement in relation to bitemporal tables is to present start and end of the valid time as well as its currentness in relation to transaction i.e. system time.

4.1 Basic Specifications

The **B_departments** table is an SQL table used for testing of bitemporal logic implemented in hand-coded applications. The CREATE TABLE statement for this table is given in Example 14. (The second CREATE TABLE statement in this example creates the corresponding history table.) Similarly, the **BITemp_departments** table is a table with the built-in temporal logic and it handles bitemporal time. As in the case of system-versioned tables, the user has to create a corresponding history table and to activate versioning for it. The corresponding CREATE TABLE statements are given in Example 15.

Example 14

```
CREATE TABLE B_departments(
    D_id INT NOT NULL, D_Name CHAR(20), D_Dept INT,
    V_Start DATE NOT NULL, V_End DATE NOT NULL, -- BUSINESS_TIME
    D_Start TIMESTAMP(12), D_End TIMESTAMP(12), -- SYSTEM_TIME
    PRIMARY KEY (D_id, V_Start));
-- Start_Time <= End_Time ->  must be explicitly checked
ALTER TABLE B_departments
    ADD CONSTRAINT B_departments_check CHECK(V_Start <= V_End);
-- History Table
CREATE TABLE B_departments_History(
    D_id INT NOT NULL, D_Name CHAR(20), D_Dept INT,
    V_Start DATE NOT NULL, V_End DATE NOT NULL,
    D_Start TIMESTAMP(12) NOT NULL, D_End TIMESTAMP(12) NOT NULL,
    PRIMARY KEY (D_id, D_Start, V_Start));
```

Example 15

```
CREATE TABLE BITemp_departments(
    D_id INT NOT NULL, D_Name CHAR(20), D_Dept INT,
    -- BUSINESS_TIME
    V_Start DATE NOT NULL, V_End DATE NOT NULL,
    PERIOD BUSINESS_TIME (V_Start, V_End),
    -- SYSTEM_TIME
    D_Start TIMESTAMP(12) GENERATED ALWAYS AS ROW BEGIN NOT NULL,
    D_End TIMESTAMP(12) GENERATED ALWAYS AS ROW END NOT NULL,
    TRANS_Start GENERATED ALWAYS AS TRANSACTION
                START ID IMPLICITLY HIDDEN,
    PERIOD SYSTEM_TIME (D_Start, D_End),
    PRIMARY KEY (D_Id, BUSINESS_TIME WITHOUT OVERLAPS));
CREATE TABLE BITemp_Dept_history LIKE BITemp_departments;
ALTER TABLE BITemp_departments
    ADD VERSIONING USE HISTORY TABLE BITemp_Dept_history;
```

The following two examples show how rows can be inserted in the **B_departments** and **BITemp_departments** tables, respectively.

Example 16

```
INSERT INTO B_Departments (D_Id, D_Name, V_Start, V_End, D_Start, D_End)
    VALUES (1, 'D2', DATE '2011-01-01', DATE '2012-01-01',
            CURRENT TIMESTAMP, DATE '9999-12-31');
```

Example 17

```
INSERT INTO BITemp_Departments (D_Id, D_Name, V_Start, V_End)
    VALUES (1, 'D1', DATE '2011-01-01', DATE '2012-01-01');
```

The INSERT statements in Examples 16 and 17 are different, because in the case of built-in temporal logic, the start system time and end system time are assigned by the system.

4.2 Key Integrity Constraints on Bitemporal Tables

The integrity constraint for primary key on bitemporal tables is similar to the corresponding constraint for application-time period tables. The INSERT statement in Example 18 violates this constraint.

Example 18

```
INSERT INTO BITemp_Departments (D_Id, D_Name, V_Start, V_End)
    VALUES (115, 'D1', DATE '2011-02-01', DATE '2012-03-01');
```

4.3 DML Statements and Bitemporal Tables

Temporal modifications for bitemporal tables are similar to temporal modifications for application-time period tables. The main difference is that updates (instead of deletions) are performed on transaction end time.

Example 19

```
DELETE FROM BITemp_departments FOR PORTION OF BUSINESS_TIME
    FROM DATE '2011-01-01' TO DATE '2011-02-01' WHERE D_id = 115;
```

For the DELETE statement in Example 19 above, the system performs two statements, an UPDATE and an INSERT. The UPDATE statement will modify the existing row. The start valid time of the row will be updated to the new value (2011-02-01), while the start system time and end system time will be the time, when the DELETE statement above has been executed and forever, respectively. (The end valid time will be unchanged.) The INSERT statement inserts a new row in the history table and modifies the system end time of the original row from forever to the time, when the row has been deleted. Both valid time values as well as the start system time will be unchanged.

5 Test Results

To evaluate performance tests described above, we use a host system with 8 GB RAM and the Intel Q6600 processor with 2.4 GHz. The software was installed on a virtual machine (VMware) with 3 GB RAM and four kernels. The operating system was Windows 7 Professional 64 Bit.

The database system used for testing is IBM DB2 V 10.5 (64 bit edition), with IBM Data Studio V 3.2 as interface to the database server. Although Data Studio displays execution times for each executed statement, our tests showed that these times are unreliable i.e. vary significantly from each other and therefore this tool could not be used for performance test. For this reason, we used a command script, which starts the **db2batch** command. Each table is consecutively loaded with 100, 1000, 10000, and 100000 rows and tests with each load have been executed ten times. The tables below contain the following columns: number of rows, the tables name, average of each execution time and corresponding standard deviation. The last column shows the ratio between execution times of built-in code and execution times of corresponding triggers. All measures are given in milliseconds.

Table 3. Application-time period table, test of PK integrity

# Rows	Table	Mean (ms)	Standard devation (ms)	Speedup
100	Employees	72	9	
100	T_employees	59	9	1.21
1000	Employees	824	68	
1000	T_employees	620	58	1.33
10000	Employees	9135	202	
10000	T_employees	6878	95	1.33
100000	Employees	90229	1397	
100000	T_employees	69661	469	1.30

Table 3 shows execution times for violation of PK integrity with the INSERT statement in Examples 5 and 6. For all loads, the ratio shows that execution time of temporal logic is ca. 1.3 times faster than corresponding hand-coded applications.

Table 4 shows execution times for the DELETE statement in Example 7 and the corresponding stored procedure. For all but first load, the ratio shows the similar results as for the previous test, i.e. that execution time of temporal logic is ca. 1.45 times faster than corresponding hand-coded applications. (The anomalous result is due to the very small amount of rows in the first load.)

Table 4. Application-time period table: Execution of a DML statement (DELETE)

# Rows	Table	Mean (ms)	Standard devation (ms)	Speedup
100	Employees	109	12	
100	T_employees	514	54	0.21
1000	Employees	1007	11	
1000	T_employees	694	92	1.45
10000	Employees	9004	273	
10000	T_employees	6089	247	1.48
100000	Employees	88388	613	
100000	T_employees	58956	223	1.50

Table 5. System-versioned table: Execution of a DML statement (UPDATE)

# Rows	Table	Mean (ms)	Standard devation (ms)	Speedup
100	Departments	79	16	
100	S_departments	63	9	1.24
1000	Departments	813	100	
1000	S_departments	616	70	1.32
10000	Departments	8033	94	
10000	S_departments	6936	178	1.16
100000	Departments	82026	2444	
100000	S_departments	70803	723	1.16

Table 6. System-versioned table: Execution of a DML statement (DELETE)

# Rows	Table	Mean (ms)	Standard devation (ms)	Speedup
100	Departments	69	10	
100	S_departments	59	8	1.18
1000	Departments	658	85	
1000	S_departments	544	78	1.21
10000	Departments	7885	164	
10000	S_departments	6597	524	1.20
100000	Departments	82987	2302	
100000	S_departments	71040	696	1.17

Table 5 shows execution times for UPDATE in Example 12 and the corresponding trigger. The execution times differ non-significantly from the times in Table 3 and 4. The slightly better ratio for hand-coded applications can be explained with the minor complexity of system-versioned tables (see Section 3.3). The same is true for execution times of the DELETE statement in Table 6.

Table 7 shows execution times for violation of PK integrity with the INSERT statement for bitemporal tables. The corresponding statement is given in Example 18 and the corresponding trigger. The semantics of this constraint

Table 7. Bitemporal table: Test of PK integrity

# Rows	Table	Mean (ms)	Standard devation (ms)	Speedup
100	B_departments	75	17	
100	BITemp_departments	59	10	1.27
1000	B_departments	895	76	
1000	BITemp_departments	668	50	1.34
10000	B_departments	10546	200	
10000	BITemp_departments	7713	146	1.37

Table 8. Bitemporal table: Execution of a DML statement (DELETE)

# Rows	Table	Mean (ms)	Standard devation (ms)	Speedup
100	B_departments	297	25	
100	BITemp_departments	58	11	5.13
1000	B_departments	3175	262	
1000	BITemp_departments	573	50	5.54
10000	B_departments	45078	1813	
10000	BITemp_departments	7277	225	6.19

for bitemporal tables is equivalent to the semantics of the same constraint for application-time period tables. Therefore, the ratio of execution times is similar.

Table 8 shows execution times for a DML statement (Example 19) for bitemporal tables. As can be seen in Section 4.3, the semantics of UPDATE and DELETE statements on bitemporal tables is very complex. (The most complex form of an UPDATE statement requires an UPDATE and three INSERT statements.) Therefore, execution times of triggers and stored procedures for DML statements of bitemporal tables are several times more slowly than the corresponding implementation of temporal logic.

6 Conclusions and Future Work

In database systems, which do not support built-in temporal logic, users have to implement this functionality in their application code. The SQL:2011 standard with its specification of temporal data and IBM DB2 with the implementation of the same dramatically simplify the code that has to be written.

In order to show execution times of built-in temporal logic vs. corresponding hand-coded applications, we performed experiments to measure the performance of both groups using the same data. Our performance tests showed three important results. First, execution times of built-in code is ca. 1.5 times quicker than execution time of the corresponding triggers and stored procedures. Second, standard deviation of hand-coded applications is significantly greater than corresponding deviation for applications implemented using built-in temporal logic. Another conclusion is: the more complex the semantics of a statement, the bigger the difference in their execution times.

It is well known that the first specification of temporal data model in SQL:2011 is non-satisfying [11]. The main defect is that the standard, at this time, does not specify two very important temporal features: the PERIOD data type and coalescing. The PERIOD data type is defined as the time interval, which contains a set of consecutive time units. This data type has the lower and upper limit, which are both of type DATE or TIMESTAMP. The most important property of this data type is that it can be used in the natural way to represent time intervals. Additionally, it supports operations, such as CONTAINS, EQUALS, PRECEDES and OVERLAPS as methods of the data type. We expect that the support of this data type will come soon, and the study of performance advantages of the PERIOD data type is one of our main goals in the future. Also, the current standard lacks the support for coalescing. Coalescing means that the system automatically merges the rows of a table, which overlaps [2]. This problem appears very often when an INSERT (UPDATE) statement is performed, and time-invariant attributes of the rows are equal and at the same time their timestamps overlaps or adjoin. It can be expected that performance gains of the support for coalescing will bring significant benefits in relation to hand-coded solution. For this reason, this is also one of the goals in our work.

References

1. Atay, C.: A Comparison of Attribute and Tuple Time Stamped Bitemporal Relational Data Models. In: Int. Conf. on Applied Computer Science (2010)
2. Boehlen, M., Snodgrass, R.: Coalescing in Temporal Databases, VLDB (1996)
3. Darwen, H., Date, C.: An Overview and Analysis of Proposals Based on the TSQL2 Approach (1996), http://www.dcs.warwick.ac.uk/~hugh/TTM/OnTSQL2.pdf (last visit: February 14, 2014)
4. Gadia, S.: Ben-Zvi's Pioneering Work in Relational Temporal Databases. In: Tansel, A., et al. (eds.) Temporal Databases. Benjamin/Cummings (1993)
5. Grandi, F.: Introducing an Annotated Bibliography on Temporal and Evolution Aspects in the Semantic Web. SIGMOD Records 41(4) (2012)
6. Kulkarni, K., Michels, J.: Temporal Features in SQL:2011. SIGMOD Records 41(3) (2012)
7. ISO/IEC 9075-2:2011: Database languages: SQL, Part 2 (2011)
8. Jensen, C.S., et al.: The consensus glossary of temporal database concepts - February 1998 version. In: Etzion, O., Jajodia, S., Sripada, S. (eds.) Temporal Databases - Research and Practice. LNCS, vol. 1399, pp. 367–405. Springer, Heidelberg (1998)
9. Lorentzos, N.: The Interval-extended Relational Model and Its Applications to Valid-time. In: Temporal Databases (1993)
10. Nicola, M., Sommerlandt, M.: Managing time in DB2 with temporal consistency. IBM Developers Works (2011)
11. Petković, D.: Was lange währt, wird endlich gut: Temporale Daten im SQL-Standard. Datenbank-Spektrum 13(2), 131–138 (2013) (in German)
12. Saracco, C., Nicola, M., Gandhi, L.: A matter of time: Temporal data management in DB2 (2012), http://www.ibm.com/developerworks/data/library/techarticle/dm-1204db2temporaldata/dm-1204db2temporaldata-pdf.pdf (last visit: February 14, 2014)

13. Snodgrass, R., Ahn, I.: Performance Evaluation of a Temporal Database Management System. Communications of ACM (1986)
14. Snodgrass, R.: The TSQL2 Temporal Query Language. Kluwer (1995)
15. Tansel, A., Clifford, J., Gadia, S., Jajodia, S., Segev, A., Snodgrass, R.: Temporal Databases (1993)
16. Toman, D.: A Point-based Temporal Extension of SQL. In: Bry, F., Ramakrishnan, R., Ramamohanarao, K. (eds.) DOOD 1997. LNCS, vol. 1341, pp. 103–121. Springer, Heidelberg (1997)
17. Bach, M., Werner, A.: Standardization of NoSQL Database Languages. In: Kozielski, S., Mrozek, D., Kasprowski, P., Małysiak-Mrozek, B., Kostrzewa, D. (eds.) BDAS 2014. CCIS, vol. 424, pp. 50–60. Springer, Heidelberg (2014)