Temporal Support for Persistent Stored Modules

Richard T. Snodgrass [*1], Dengfeng Gao [#2], Rui Zhang [*3], and Stephen W. Thomas [†4]

[*] *University of Arizona, Tucson, AZ USA* [1] rts@cs.arizona.edu [3] ruizhang@cs.arizona.edu

[#] *IBM Silicon Valley Lab, San Jose, CA USA* [2] dgao@us.ibm.com

[†] *Queen's University, Kingston, ON Canada* [4] sthomas@cs.queensu.ca

***Abstract*—We show how to extend temporal support of SQL to the Turing-complete portion of SQL, that of persistent stored modules (PSM). Our approach requires minor new syntax beyond that already in SQL/Temporal to define and to invoke PSM procedures and functions, thereby extending the current, sequenced, and non-sequenced semantics of queries to such routines. Temporal upward compatibility (existing applications work as before when one or more tables are rendered temporal) is ensured. We provide a transformation that converts Temporal SQL/PSM to conventional SQL/PSM. To support sequenced evaluation of stored functions and procedures, we define two different slicing approaches, *maximal slicing* and *per-statement slicing*. We compare these approaches empirically using a comprehensive benchmark and provide a heuristic for choosing between them.**

I. Introduction

Temporal query languages are now fairly well understood, as indicated by 80-some encyclopedia entries on various aspects of time in databases and query languages [1] and through support in prominent DBMSes. Procedures and functions in the form of *Persistent Stored Modules* (PSM) have been included in the SQL standard and implemented in numerous DBMSes [2]. However, no work to date has appeared on the combination of stored procedures and temporal data.

The SQL standard includes stored routines in Part 4: control statements and persistent stored modules (PSM) [3]. Although each commercial DBMS has its own idiosyncratic syntax and semantics, stored routines are widely available in DBMSes and are used often in database applications, for several reasons. Stored routines provide the ability to compile and optimize SQL statements and the corresponding database operations once and then execute them many times on demand, within the DBMS and thus close to the data. This represents a significant reduction in resource utilization and savings in the time required to execute those statements. The computational completeness of the language enables complex calculations and allows users to share common functionality and encourage code reuse, thus reducing development time [2].

It has been shown that queries on temporal data are often hard to express in conventional SQL: the average temporal query/modification is three times longer in terms of lines of SQL than its nontemporal equivalent [4]. There have been a large number of temporal query languages proposed in the literature [1], [5], [6], [7]. Previous change proposals [8], [9] for the SQL/Temporal component of the SQL standard showed how SQL could be extended to add temporal support while guaranteeing that the new temporal query language was compatible with conventional SQL. That effort is now moving into commercial DBMSes. Oracle 10g added support for valid-time tables, transaction-time tables, bitemporal tables, sequenced primary keys, sequenced uniqueness, sequenced referential integrity, and sequenced selection and projection, in a manner quite similar to that proposed in SQL/Temporal. Oracle 11g enhanced support for valid-time queries [10]. Teradata recently announced support in Teradata Database 13.10 of most of these facilities as well [11], as did IBM for DB2 10 for z/OS [12]. These DBMSes all support PSM, but not invocation of stored routines within sequenced temporal queries. For completeness and ease of use, temporal SQL should include stored modules.

The problem addressed by this paper is thus quite relevant: how can SQL/PSM be extended to support temporal relations, while easing migration of legacy database applications and enabling complex queries and modifications to be expressed in a consistent fashion? Addressing this problem will enable vendors to further their implementation of temporal SQL.

In this paper, we introduce minimal syntax that will enable PSM to apply to temporal relations; we term this new language Temporal SQL/PSM. We then show how to transform such routines in a source-to-source conversion into conventional PSM. Transforming sequenced queries turn out to be the most challenging. We identify the critical issue of supporting sequenced queries (in any query language), that of time-slicing the input data while retaining period timestamping. We then define two different slicing approaches, *maximally-fragmented slicing* and *per-statement slicing*. The former accommodates the full range of PSM statements, functions, and procedures in temporal statements in a minimally-invasive manner. The latter is more complex, supports almost all temporal functions and procedures, utilizes relevant compile time analysis, and often provides a significant performance benefit, as demonstrated by an empirical comparison using DB2 on a wide range of queries, functions, procedures, and data characteristics.

To our knowledge, this is the first paper to propose temporal syntax for PSM, the first to show how such temporally enhanced queries, functions, and procedures can be implemented, and the first to provide a detailed performance evaluation.

II. SQL/PSM

Persistent stored modules (PSM) are compiled and stored in the schema, then later run within the DBMS. PSM consists of *stored procedures* and *stored functions*, which are collectively called *stored routines*. Stored routines can be written in

```
CREATE FUNCTION get_author_name (aid CHAR(10))
  RETURNS CHAR(50)
  READS SQL DATA
  LANGUAGE SQL
  BEGIN
    DECLARE fname CHAR(50);
    SET fname = (SELECT first_name
                 FROM author
                 WHERE author_id = aid);
    RETURN fname;
  END;
```

Fig. 1. PSM function `get_author_name()`

```
SELECT i.title
FROM item i, item_author ia
WHERE i.id = ia.item_id
      AND get_author_name(ia.author_i) = 'Ben';
```

Fig. 2. An SQL query calling `get_author_name()`

either SQL or one of the programming languages with which SQL has defined a binding (such as Ada, C, COBOL, and Fortran). Stored routines written entirely in SQL are called *SQL routines*; stored routines written in other programming languages are called *external routines*.

As mentioned above, each commercial DBMS has its own idiosyncratic syntax and semantics of PSM. For example, the language PL/SQL used in Oracle supports PSM and control statements. Microsoft's Transact-SQL (similar to Sybase's) provides extensions to standard SQL that permit control statements and stored procedures. IBM, MySQL, Oracle, PostgreSQL, and Teradata all have their own implementation of features similar to those in SQL/PSM.

We'll use a running example through the paper of a stored routine written in SQL and invoked in a query. This example is from a bookstore application with tables `item` (that is, a book) and `publisher`. In Figure 1, the conventional (non-temporal) stored function `get_author_name()` takes a book author ID as input and returns the first name of the author with that ID. The SQL query in Figure 2 returns the title of the item that has a matching author whose first name is Ben. This query calls the function in its where clause. Of course, this query can be written without utilizing stored functions; our objective here is to show how a stored routine can be used to accomplish the task.

III. SQL/TEMPORAL

SQL/Temporal [8], [9] was proposed as a part of the SQL:1999 standard [3]. Many of the facilities of this proposal have been incorporated into commercial DBMSes, specifically IBM DB2 10 for z/OS, Oracle 11g and Teradata 13.10. Hence, SQL/Temporal is an appropriate language definition for considering temporal support of stored routines. In the context of databases, two time dimensions are of general interest: valid time and transaction time [13]. In this paper, we focus on valid time, but everything also applies to transaction time. (Previous work by the authors on temporal query language implementation has shown that the combination of valid and transaction time to bitemporal tables and queries is straightforward, but the details of supporting bitemporal data in the PSM transformations to be discussed later have not yet been investigated.)

We have identified two important features that provide easy migration for legacy database applications to temporal systems: *upward compatibility (UC)* and *temporal upward compatibility (TUC)* [14]. Upward compatibility guarantees that the existing applications running on top of the temporal system will behave exactly the same as when they run on the legacy system. Temporal upward compatibility ensures that when an existing database is transformed into a temporal database, legacy queries still apply to the current state.

To ensure upward compatibility and temporal upward compatibility [14], SQL/Temporal classifies temporal queries into three categories: *current queries*, *sequenced queries*, and *nonsequenced queries* [8]. Current queries only apply to the current state of the database. Sequenced queries apply independently to each state of the database over a specified temporal period. Users don't need to explicitly manipulate the timestamps of the data when writing either current queries or sequenced queries. Nonsequenced queries are those temporal queries that are not in the first two categories. Users explicitly manipulate the timestamps of the data when writing nonsequenced queries.

Two additional keywords are used in SQL/Temporal to differentiate the three kinds of queries from each other. Queries without temporal keywords are considered to be current queries; this ensures temporal upward compatibility [14]. Hence, the query in Figure 2 is a perfectly reasonable current query when one or more of the underlying tables is time-varying. Suppose that the `item`, `author`, and `item_author` tables mentioned above are now all temporal tables with valid-time support. That is, each row of each table is associated with a valid-time period. As before, the semantics of this query is, "list the title of the item that (currently) has a matching author whose (current) first name is Ben."

Sequenced and nonsequenced queries are signaled with the temporal keywords `VALIDTIME` and `NONSEQUENCED VALIDTIME`, respectively, in front of the conventional queries. The latter in front of the SQL query in Figure 2 requests "the title of items that (at any time) had a matching author whose first name (at any—possibly different—time) was Ben." These keywords modify the semantics of the entire SQL statement (whether a query, a modification, a view definition, a cursor, etc.) following them; hence, these keywords are termed *temporal statement modifiers* [15].

The sequenced modifier (`VALIDTIME`) is the most interesting. A query asking for "the *history* of the title of the item that has a matching author whose first name is Ben" could be written as the sequenced query in Figure 3. It is important to understand the semantics of this query. (Ignore for now that this query invokes a stored function. Our discussion here is general.) Effectively the query after the modifier (which is just the query of Figure 2) is invoked at every time granule (in this case, every day, assuming a valid-time granularity of `DATE`) over the entire time line, *independently*. So the query of Figure 2 is evaluated for January 1, 2010, using the rows

```
VALIDTIME SELECT i.title
FROM item i, item_author ia
WHERE i.id = ia.item_id
     AND get_author_name(ia.author_id) = 'Ben';
```

Fig. 3. A sequenced query calling get_author_name()

```
SELECT i.title,
     LAST_INSTANCE(i.begin_time,ia.begin_time),
     FIRST_INSTANCE(i.end_time,ia.end_time)
FROM item i, item_author ia
WHERE i.id = ia.item_id
  AND get_author_name(ia.author_i) = 'Ben'
  AND LAST_INSTANCE(i.begin_time,ia.begin_time)
      < FIRST_INSTANCE(i.end_time,ia.end_time);
```

Fig. 4. The transformed query corresponding to Figure 3 (note: incomplete)

valid on that day in the item and item_author tables, to evaluate a result for that day. The query is then evaluated for January 2, 2010, using the rows valid on *that* day, and so forth. The challenge is to arrive at this semantics via manipulations on the *period* timestamps of the data.

A variant of a sequenced modifier includes a specific period (termed the *temporal context*) such as the year 2010 after the keyword, restricting the result to be within that period.

One approach to the implementation of SQL/Temporal is to use a *stratum*, a layer above the query evaluator that transforms a temporal query defined on temporal table(s) into a (generally more complex) conventional SQL query operating on conventional tables with additional timestamp columns [16]. Implementing nonsequenced queries in the stratum is trivial. Current queries are special cases of sequenced queries. SQL/Temporal defined temporal algebra operators for sequenced queries [8]. When the stratum receives a temporal query, it is first transformed into temporal algebra, then into the conventional algebra, and finally into conventional SQL. Hence, the sequenced query of Figure 3 (again, ignoring the function invocation for the moment) would be transformed into the conventional query shown in Figure 4. This query uses a temporal join. The semantics of joins operating independently on each day is achieved by taking the intersection of the validity periods. (Note that FIRST_INSTANCE() and LAST_INSTANCE() are stored functions, defined elsewhere, that return the earlier or later, respectively, of the two argument times.) Other SQL constructs, such as aggregates and subqueries, can also be transformed, manipulating the underlying validity periods to effect this illusion of evaluating the entire query independently on each day.

While SQL/Temporal extended the data definition statements and data manipulation statements in SQL, it never mentioned PSM. The central issue before us is how to extend PSM in a coherent and consistent fashion so that temporal upward compatibility is ensured and that the full functionality of PSM can be applied to tables with valid-time and transaction-time support. Specifically, what should be done with the invocation of the stored function get_author_name(), a function that itself references the (now temporal) table item_author? What syntactic changes are needed to PSM to support time-varying data? What semantic changes are needed? How can Temporal SQL/PSM be implemented in an efficient manner? Does the stratum approach even work in this case? What optimizations can be applied to render a more efficient implementation? In this paper, we will address all of these questions.

IV. SQL/Temporal and PSM

In this section, we first define the syntax and semantics of Temporal SQL/PSM informally, provide the formal structure for a transformation to conventional SQL/PSM, then consider current queries. We then turn to sequenced queries.

A. Motivation and Intuition

We considered three approaches to extend stored routines, discussed elsewhere [17]. The basic role of a DBMS is to move oft-used data manipulation functionality from a user-developed program, where it must be implemented anew for each application, into the DBMS. In doing so, this functionality need be implemented once, with attendant efficiency benefits. This general stance favors having the semantics of a stored routine to be implied by the *context* of that invocation. Hence, for example, the temporal modifier of the SQL query that invoked a stored function would specify the semantics of that invocation. This approach assigns the most burden to the DBMS implementor and imposes the least burden on the application programmer.

As an example, the conventional query in Figure 2 will be acceptable whether or not the underlying tables are time-varying. Say that all three tables have valid-time support. In that case, this query requests the title of the item that *currently* has a matching author whose first name is Ben. (This is the same semantics that query had before, when the tables were not temporal, instead stating just the current information. This is exactly the highly valuable property of temporal upward compatibility [14].)

If we wish the *history* of those titles over time, as Ben authors more books, we would use the query in Figure 3, which employs the temporal modifier VALIDTIME. This modifies the entire query, and thus the invocation of the stored function get_author_name(). Conceptually, this function is invoked for every day, potentially resulting in different results for different authors and for different days. (Essentially, the result for a particular author_id will be time-varying, with a first name string value for each day.)

What this means is that there are no syntax extensions required to effect the current, sequenced, and non-sequenced semantics of queries (and modifications, views, etc.) that invoke a stored function. Upward compatibility (existing applications work as before) and temporal upward compatibility (existing applications work as before when one or more tables are rendered temporal) are both ensured.

Since a stored routine can be invoked from another such routine, it is natural for the context to also be retained. This implies that a query *within* a stored routine should normally *not* have a temporal modifier, as the context provides the semantics. (For example, a query within a stored routine called from a sequenced query would necessarily also be sequenced.) This feature of stored routines eases the reuse

of existing modules written in conventional SQL. But what if the user specifies a temporal modifier on a query within a stored routine? In that case, that routine can only be invoked within a nonsequenced context, which assumes that the user is manually managing the validity periods. So it is perfectly fine for the user to specify, e.g., VALIDTIME within a stored routine, but then that routine will generate a semantic error when invoked from anything but a non-sequenced query.

B. Formal Semantics

We now define the formal syntax and semantics of temporal SQL/PSM query expressions. The formal syntax is specified in conventional BNF. The semantics is defined in terms of a transformation from Temporal SQL/PSM to conventional SQL/PSM. While this source-to-source transformation would be implemented in a stratum within the DBMS, we specify this transformation using a syntax-directed denotational semantics style formalism [18] to specify the transformation from temporal SQL/PSM to conventional SQL/PSM. Such semantic functions each take a syntax sequence (with terminals and nonterminals) and transform that sequence into a string, often calling other semantic functions on the non-terminals from the original syntax sequence.

In SQL/Temporal, there are three kinds of SQL queries in which PSMs can be invoked. The production of a temporal query expression can be written as follows.

$$\langle \text{Temporal Q} \rangle \quad ::= \quad (\ \texttt{VALIDTIME}\ (\texttt{[}\langle \text{BT} \rangle\texttt{,}\ \langle \text{ET} \rangle\texttt{]})^{?} |\ \texttt{NONSEQUENCED VALIDTIME}\ )^{?} \langle \text{Q} \rangle$$

In this syntax, the question marks denote optional clauses. $\langle \text{Q} \rangle$ is a conventional SQL query. $\langle \text{BT} \rangle$ and $\langle \text{ET} \rangle$ are the beginning and ending times of the query, respectively, if it is sequenced. A query in SQL/Temporal is a current query by default (that is, without the temporal keyword(s)), or a sequenced query if the keyword VALIDTIME is used, or a nonsequenced query if the keyword NONSEQUENCED VALIDTIME is used. Note that $\langle \text{Q} \rangle$ may invoke one or more stored functions. The semantics of $\langle \text{Temporal Q} \rangle$ is expressed with the semantic function $\mathit{TSQLPSM}[\![]\!]$. $\mathit{cur}[\![]\!]$, $\mathit{seq}[\![]\!]$, and $\mathit{nonseq}[\![]\!]$ are the semantic functions for current queries, sequenced queries, and nonsequenced queries, respectively. The traditional SQL semantics is represented by the semantic function $\mathit{SQL}[\![]\!]$; this semantic function just emits its argument literally, in a recursive descent pass over the parse tree. (We could express this in denotational semantics with definitions such as

$$\mathit{SQL}[\![\texttt{SELECT}\ \langle \text{Q} \rangle ...]\!] \quad = \quad \texttt{SELECT}\ \mathit{SQL}[\![\langle \text{Q} \rangle]\!] ...$$

but will omit such obvious semantic functions that mirror the BNF productions.)

$$\begin{aligned} \mathit{TSQLPSM}[\![\langle \text{Q} \rangle]\!] &= \mathit{cur}[\![\langle \text{Q} \rangle]\!] \\ \mathit{TSQLPSM}[\![\texttt{VALIDTIME [}\langle \text{BT} \rangle\texttt{,}\ \langle \text{ET} \rangle\texttt{]}\ \langle \text{Q} \rangle]\!] & \\ &= \mathit{seq}[\![\langle \text{Q} \rangle]\!]\ \texttt{[}\langle \text{BT} \rangle\texttt{,}\ \langle \text{ET} \rangle\texttt{]} \\ \mathit{TSQLPSM}[\![\texttt{NONSEQUENCED VALIDTIME}\ \langle \text{Q} \rangle]\!] & \\ &= \mathit{nonseq}[\![\langle \text{Q} \rangle]\!] \end{aligned}$$

SQL/Temporal proposed definitions for the $\mathit{cur}[\![]\!]$ and $\mathit{seq}[\![]\!]$ semantic functions [8], [9] used above. The temporal relational algebra defined for temporal data statements cannot express the semantics of control statements and stored routines. Therefore, we need to use different techniques.

We first show how to transform current queries, then present two techniques transforming sequenced queries, namely, *maximally-fragmented slicing* and *per-statement slicing*. Nonsequenced queries require only renaming of timestamp columns and so will not be presented here.

C. Current Semantics

The semantics of a current query on a temporal database is exactly the same as the semantics of a regular SQL query on the current timeslice of the temporal database. The formal semantics of current query can be defined as taking the existing SQL semantics followed by an additional predicate.

$$\mathit{cur}[\![\langle \text{Q} \rangle]\!]\ (r_1, r_2, ..., r_n) \quad = \quad \mathit{SQL}[\![\langle \text{Q} \rangle]\!]\ \tau^{vt}_{now}(r_1, r_2, ..., r_n)$$

In this transformation, r_1, r_2, ..., r_n denote tables that are accessed by the query $\langle \text{Q} \rangle$. We borrow the temporal operator τ^{vt}_{now} from the proposal of SQL/Temporal [9]. τ^{vt}_{now} extracts the current timeslice value from one (or more) tables with valid-time support.

Calculating the current timeslice of a table is equivalent to performing a selection on the table. To transform a current query (with PSM) in SQL, we just need to add one predicate for each table to the where clauses of the query and queries inside the PSM. Assume r_1, ..., r_n are all the tables that are accessed by the current query. The following predicate needs to be added to *all* the where clauses whose associated from clause mentions a temporal table.

```
r1.begin_time <= CURRENT_TIME AND
   r1.end_time > CURRENT_TIME AND
...
rn.begin_time <= CURRENT_TIME AND
   rn.end_time > CURRENT_TIME
```

As an example, the current version of the function in Figure 1 should be transformed to the SQL query in Figure 5 and the current query in Figure 2 is transformed to the SQL query in Figure 6.

V. Maximally-Fragmented Slicing

Maximally-fragmented slicing applies small, isolated changes to the routines by adding simple predicates to the SQL statements inside the routines to support sequenced queries. The idea of maximally-fragmented slicing is similar to that used to define the semantics of τXQuery queries [19], which adapted the idea of *constant periods* originally introduced to evaluate (sequenced) temporal aggregates [20].

The basic idea is to first collect at compile time all the temporal tables that are referenced directly or indirectly by the query, then compute all the *constant periods* over which the result will definitely not change, and then independently evaluate the routine (and any routines invoked indirectly) for

```
CREATE FUNCTION curr_get_author_name
                          (aid CHAR(10))
  RETURNS CHAR(50)
  READS SQL DATA
  LANGUAGE SQL
  BEGIN
    DECLARE fname CHAR(50);
    SET fname = (SELECT first_name
         FROM author
         WHERE author_id = aid
           AND author.begin_time <= CURRENT_TIME
           AND author.end_time > CURRENT_TIME);
    RETURN fname;
  END;
```

Fig. 5. Current version of the function in Figure 1

```
SELECT i.title
FROM item i, item_author ia
WHERE i.id = ia.item_id
  AND curr_get_author_name(ia.author_i) = 'Ben'
  AND i.begin_time <= CURRENT_TIME
  AND i.end_time > CURRENT_TIME
  AND ia.begin_time <= CURRENT_TIME
  AND ia.end_time > CURRENT_TIME;
```

Fig. 6. The SQL query transformed from Figure 2

each constant period, with one result for each such identified period. We compute the constant periods first, then the SQL query is evaluated at each constant period.

Figure 7(a) provides an example of periods from the three tables. (Here, time advances to the right.) Figure 7(b) shows the result of computing the nine constant periods; the asterisks show the nine calls to the stored function.

In the remainder of this section, we first describe the transformation of queries that invoke functions; we then consider nested invocation. Then we will consider external routines.

A. Constant Periods

A sequenced query is conceptually evaluated at every time instant, e.g., every day. But we can make this much more efficient by finding those periods, those consecutive instants, in which the value of a stored function will not change. A *constant period* is one in which none of the tables accessed anywhere in the function's evaluation changes. Invoking the stored function for any of the time instants within a constant period is guaranteed to yield the same result.

To compute constant periods, all the timestamps in the input temporal tables are collected and the begin time and end time of each timestamp are put into a list. These time points are the only modification points of the temporal data, and thus, of the result. Instead of storing the constant periods as a sequence of elements, we take advantage of the relational model and put the constant periods into a table, each row of which has two attributes indicating the begin time and end time of a period.

Assume the tables involved in a query (including the tables appearing in the routines called by the query, which can be determined by a static analysis) are r_1, r_2, ..., r_n. The table *cp* that stores the constant periods of the n tables is calculated by the following relational calculus expression.

$$cp(r_1, r_2, ..., r_n) = \{\langle bt, et\rangle | bt \in ts \wedge et \in ts \wedge bt < et \wedge \neg\exists t \in ts(bt < t < et)\}$$

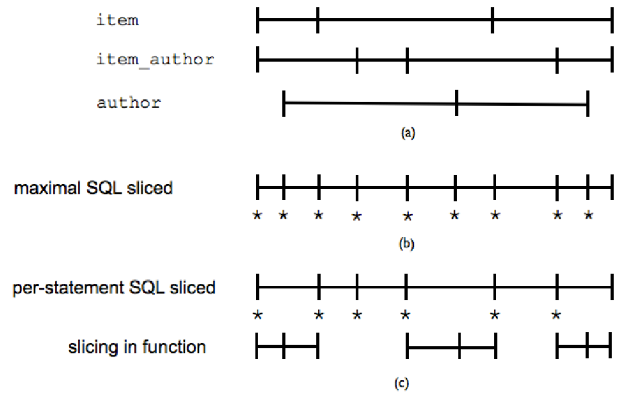

Fig. 7. Comparison of Slicing

```
CREATE TEMPORARY TABLE TS AS (
  SELECT begin_time AS time_point FROM author
  UNION
  SELECT end_time AS time_point FROM author
  UNION
  SELECT begin_time AS time_point FROM item
  UNION
  SELECT end_time AS time_point FROM item
  UNION
  SELECT begin_time AS time_point
  FROM item_author
  UNION
  SELECT end_time AS time_point
  FROM item_author);

CREATE VIEW CP AS (
SELECT ts1.time_point AS begin_time,
         ts2.time_point AS end_time
FROM ts AS ts1, ts AS ts2
WHERE begin_time < end_time AND
   min_time <= ts1.time_point AND
   ts1.time_point < max_time AND
   NOT EXISTS (SELECT time_point
               FROM ts
               WHERE begin_time < time_point
                  AND time_point < end_time)
ORDER BY begin_time;
```

Fig. 8. Computing the constant periods

This can be evaluated by a self-join of the single column table ts, which stores all the begin time and end time of the timestamps in the n input temporal tables. The table *cp* can be obtained by running the SQL query in Figure 8. (Here the `min_time` and `max_time` delimit the temporal context mentioned in the sequenced query.) Once *cp* is computed, the original query is evaluated against the timeslice of the input tables at the begin time of each tuple in *cp*.

B. SQL Queries Invoking Functions

An SQL query is written as one or more SQL select statements. Assume a stored function `F()` is called in this select statement. The function can appear in several places. It could appear as one of the columns in the select list or as a part of the search condition specified in either the where clause or the having clause. It could also appear in the table reference list if its return value is a collection type (that is, an array, which can be used as a derived table). In any case, the sequenced semantics of the select statement is as follows. We use the semantic function *max*$[\![]\!]$ to show the transformation of maximally-fragmented slicing

max ⟦⟨select statement⟩⟧ =
  SELECT max ⟦⟨select list⟩⟧, cp.begin_time,
    cp.end_time
  FROM max ⟦⟨table reference list⟩⟧, cp
  [WHERE max ⟦⟨search condition⟩⟧ AND
  overlap ⟦*tables* ⟦⟨select statement⟩⟧⟧, cp.begin_time]
  [max ⟦⟨group by clause⟩⟧, cp.begin_time]
  [HAVING max ⟦⟨search condition⟩⟧]

A sequenced query always returns a temporal table, i.e., each row of the table is timestamped. Therefore, cp.begin_time and cp.end_time are added to the select list and cp is added to the from clause. A search condition is added to the where clause to ensure that tuples from every table overlaps the beginning of the constant period. (By definition, no table will change *during* a constant period, so checking overlaps with the start of the constant period, which is quicker than the more general overlaps, is sufficient.) The semantic function *tables* ⟦ ⟧ returns an array of strings, each is a table reference appearing in the input query. The semantic function *overlap* ⟦ ⟧ returns a series of search conditions represented as a string. If there are n tables referenced in the statement, *overlap* ⟦ ⟧ returns n conditions, each of the form

  $tname$.begin_time <= cp.begin_time AND
  cp.begin_time < $tname$.end_time

where $tname$ is the table name.

In the above transformation, four nonterminals need to be transformed: (⟨select list⟩, ⟨table reference list⟩, and two ⟨search condition⟩s). If the select statement is not a nested query, the only aspect that need to be transformed in these four nonterminals are the function calls that occur in these nonterminals, using the *maxf* ⟦ ⟧ semantic function.

max ⟦⟨function call⟩⟧ = max_⟨function name⟩ (
    maxf ⟦⟨parameter list⟩⟧)
maxf ⟦⟨parameter list⟩⟧ = ⟨parameter list⟩, begin_time

C. Stored Functions Invoked in SQL Queries

The body of the definition of F also needs to be transformed. All the SQL queries inside F are transformed in max_F to a temporal query at the input time point cp.begin_time. This transformation is done by adding a condition of overlapping cp for each temporal table in the where clause.

max ⟦⟨function definition⟩⟧ = *maxf* ⟦⟨function definition⟩⟧
maxf ⟦⟨search condition⟩⟧ = ⟨search condition⟩ AND
    overlap ⟦*tables* ⟦⟨select statement⟩⟧, begin_time⟧

The select statement could be a nested query having a subquery in either the from clause or the where clause. In this case, the subquery (a select statement) should be transformed to a temporal query at the time point cp.begin_time. Therefore, the transformation of a subquery is similar to the transformation of a query inside a function.

If another function is called inside a function, the constant period passed into the original function is also passed into the nested function. If a procedure is invoked inside the original function, the same constant period is passed to the procedure. The output parameters of the procedure remain unchanged.

As a useful optimization, if the function does not involve any temporal data, then cp.begin_time does not need to be passed as a parameter. Again, compile-time reachability analysis can propagate such an optimization.

As an example, let's look at our sequenced query in Figure 3. This query is transformated into the SQL query shown in Figure 9, which calls the function in Figure 10.

```
SELECT i.title, cp.begin_time, cp.end_time
FROM item i, item_author ia, cp
WHERE i.id = ia.item_id
  AND max_get_author_name(ia.author_i,
                         cp.begin_time) = 'Ben'
  AND i.begin_time <= cp.begin_time
  AND cp.begin_time < i.end_time
  AND ia.begin_time <= cp.begin_time
  AND cp.begin_time < ia.end_time;
```

Fig. 9. Figure 3 using maximally-fragmented slicing

```
CREATE FUNCTION max_get_author_name
                (aid CHAR(10),
                 begin_time_in DATE)
  RETURNS CHAR(50)
  READS SQL DATA
  LANGUAGE SQL
  BEGIN
    DECLARE fname CHAR(50);
    SET fname = (SELECT first_name
      FROM author
      WHERE author_id = aid
        AND author.begin_time <= begin_time_in
        AND begin_time_in < author.end_time);
    RETURN fname;
  END;
```

Fig. 10. Figure 1 using maximally-fragmented slicing

D. External Routines

For external routines written in other programming languages such as C/C++ and Java, the same transformation applies to the source code of the routine. When the external routine has SQL data manipulation statements, its source code is usually available to DBMS for precompiling and thus the transformation can be performed. In the case that a PSM is a compiled external routine, the routine must not access any database tables, and thus there is no need to transform it.

VI. Per-Statement Slicing

Maximally-fragmented slicing evaluates a stored routine many times when the base tables change frequently over time, each time at a single point in time. We therefore developed a second transformation approach, termed *per-statement slicing*, which separately slices each construct that references a temporal result, whether it be a temporal table or the result of an routine that ultimately references a temporal table. The idea of per-statement slicing is to transform each sequenced routine into a semantically-equivalent conventional routine that operates on temporal tables. Therefore, each SQL control statement inside the routines should also operate on temporal tables. This transformation produces more complex code, but that code only iterates over the partial slicing to that point.

Figure 7(c) shows the slicing that would be done in the SQL statement (between `item` and `item_author`), with six calls (the asterisks, fewer than maximal slicing) to the `get_author_name()` function. Three of those calls require further slicing on the `author` relation within the function.

We illustrate the per-statement transformation on the `get_author_name()` function and then briefly summarize more complex transformations. Recall from Figure 1 that this function consists of a function signature, a declaration of the `fname` variable, a `SET` statement, and a `RETURN` statement. Each of these constructs is transformed separately. We use the semantic function *ps* ⟦ ⟧ to show the transformation of per-statement slicing. p is an input parameter of the semantic function indicating the period of validity of the return data of the input query.

A. The Function Signature

In per-statement slicing, each routine being invoked in a sequenced query has the sequenced semantics. Hence the output and return values are all temporal tables. This requires the signature of the routine to be changed. Each sequenced function is evaluated for a particular temporal period and the return value of the sequenced function is a temporal table over that temporal period. Therefore, a temporal period is added to the input parameter list. The return value is a sequence of return values, each associated with a valid-time period. The formal transformation of a function definition is as follows.

The nonterminal ⟨function specification⟩ defines the signature of the function and includes three non-terminals, namely ⟨routine name⟩, ⟨declaration list⟩, and ⟨returns clause⟩. The transformation differentiates the name of the sequenced function from the original function with current semantics with a prefix.

ps ⟦⟨routine name⟩⟧ = ps_⟨routine name⟩

While maximally-fragmented slicing adds only a single input parameter (the begin time of the constant period), per-statement slicing adds two input parameters (the begin and end times of the period itself, named to differentiate from the *returned* periods).

ps ⟦⟨parameter declaration list⟩⟧ =
⟨parameter declaration list⟩, `min_time DATE,`
`max_time DATE`

The ⟨returns clause⟩ has the following syntax.

⟨returns clause⟩ ::= `RETURNS` ⟨data type⟩

The data type of the return value is transformed to a temporal table derived by a *collection type*. A collection type is a set of rows that have the same data structure.

ps ⟦⟨return clause⟩⟧ =
`RETURNS ROW(taupsm_result` ⟨data type⟩`,`
`begin_time DATE,`
`end_time DATE) ARRAY`

This returned temporal table is then joined with other temporal tables in the invoking query. We can then integrate the result of this function in a way very similar to that shown in Figure 3, with the only change being the use of both `begin_time` and `end_time`.

B. The Function Body

The returned value is always a temporal table (the array of rows just stated). We need to add to the function's declaration list a declaration of this table.

ps ⟦⟨decl list⟩⟧ =
⟨decl list⟩
`DECLARE psm_return`
`ROW(taupsm_result`⟨data type⟩`,`
`begin_time DATE,`
`end_time DATE) ARRAY`

We now turn to the body of the `get_author_name()` function. The first statement declares the `fname` variable, which must now be time-varying. The second statement sets the value of `fname` to the result of a select statement, which must be transformed to its sequenced equivalent. The third statement returns this variable. We employ a compile-time optimization that aliases the `fname` variable to the return variable, so that we can use the same temporal table for both.

⟨assignment statement⟩ ::= `SET` ⟨assignment target⟩ =
⟨value expression⟩

The ⟨assignment target⟩ is usually a variable. A variable inside a routine is transformed to a temporal table. Therefore a sequenced assignment statement tries to insert tuples into or update the temporal table for a certain period. Intuitively the assignment statement should be transformed to a sequenced insert or update. Here we transform it to a sequenced delete followed by an insert to the target temporal table. If there are tuples valid in the input time period, they are first deleted, then new tuples are inserted. It is the same as sequenced update. If there are no tuples valid in the input time period, a new tuple is inserted. The inserted tuples are returned from the sequenced ⟨value expression⟩.

ps ⟦⟨assignment statement⟩⟧ =
ps ⟦`DELETE FROM TABLE` ⟨assignment target⟩⟧ p;
`INSERT INTO TABLE` ⟨assignment target⟩
ps ⟦⟨value expression⟩⟧

In our example, we don't need a deletion statement in the transformed code because this is the first assignment to that variable. The transformation of the SELECT is simple, because it only contains selection (the where clause) and projection (the select clause). However, we only want the values valid within the period passed to the function.

The final statement is the return statement. Each ⟨return statement⟩ is transformed to an INSERT statement that inserts some tuples into the temporal table that stores all the return values. At the end of the function, one ⟨return statement⟩ is added to return the temporal table. The invoking query will then get the return value and use it as a temporal table.

ps ⟦⟨return statement⟩⟧ =
`INSERT INTO TABLE ps_return_tb`
ps ⟦⟨value expression⟩⟧

The sequenced ⟨value expression⟩ returns a temporal table that has three columns: one value with the same type of the ⟨value expression⟩, one `begin_time`, and one `end_time` of the valid-time period of the value. ⟨value expression⟩ could be a literal, a variable, a select statement that returns a single

```
CREATE FUNCTION ps_get_author_name(
                          aid CHAR(10),
                          min_time DATE,
                          max_time DATE)
  RETURNS ROW (taupsm_result CHAR(50),
             begin_time DATE,
             end_time DATE) ARRAY
  READS SQL DATA
  LANGUAGE SQL
  BEGIN
    DECLARE psm_result
                ROW (taupsm_result CHAR(50),
                     begin_time DATE,
                     end_time DATE) ARRAY;
    INSERT INTO psm_result
      SELECT a.first_name,
         LAST_INSTANCE(a.begin_time, min_time),
         FIRST_INSTANCE(a.end_time, max_time),
      FROM author a
      WHERE a.author_id = aid AND
         LAST_INSTANCE(a.begin_time, min_time)
           < FIRST_INSTANCE(a.end_time, max_time)
    RETURN psm_result;
  END;

SELECT i.title,
    LAST_INSTANCE(
      LAST_INSTANCE(i.begin_time,ia.begin_time),
      t.begin_time) as begin_time,
    FIRST_INSTANCE(
       FIRST_INSTANCE(i.end_time,ia.end_time),
       t.end_time) as end_time
FROM item i, item_author ia,
    ps_get_author_name(ia.author_id,
      LAST_INSTANCE(i.begin_time,ia.begin_time),
      FIRST_INSTANCE(i.end_time,ia.end_time)) t
WHERE i.id = ia.item_id AND
    t.taupsm_result = 'Ben' AND
    LAST_INSTANCE(
      LAST_INSTANCE(i.begin_time,ia.begin_time),
      t.begin_time)
     < FIRST_INSTANCE(
       FIRST_INSTANCE(i.end_time,ia.end_time),
       t.end_time)
```

Fig. 11. Per-statement transformation for Figure 3

value, or a function that returns a single value. It is trivial to transform a literal into a temporal tuple: we just need to add the valid period for the literal. A variable is transformed to a select statement that retrieves the tuples from the temporal table (the sequenced variable). The transformation of the sequenced select statement is given in previous research [9], and the transformation of a sequenced function call was defined above.

In this case, we are returning just a single variable, a variable that has been aliased to the return value already. So we just have to return that temporal table.

The result of transforming both the function and its invocation within SQL is shown in Figure 11. It is interesting to compare this result with that for maximal slicing (Figures 8, 10, and 9). In maximal slicing, we need to first do all the work of computing the (potentially many) constant periods, but then things are pretty easy from there on out: the transformed function needs to evaluate only within a constant period, where things are by definition not time-varying. In per statement slicing, on the other hand, the function caller states a somewhat restricted evaluation period, and the function itself further slices, in this case within the SELECT on the `author` periods, within the evaluation period.

C. Transforming Other SQL Statements

Denotational semantics for the transformations of all of the statements are given elsewhere [17]. Transformations for the signature, assignment statement, and return statement were discussed above in some detail. We now end with a summary of the other statements.

As befits its name, per-statement slicing will slice on time whenever a time-varying relation is involved, either directly or indirectly as the return value of a function call or SQL statement or through a time-varying value in a variable. (Indeed, a time-varying relation is generally encountered through a time-varying variable such as a cursor, so all of the alternatives come down to time-varying variables.) As PSM is a block-structured language, slicing is also block structured. Compile-time analysis is used determine the scope of each time-varying variable. Upon encountering such a variable, the transformation inserts a WHILE loop that iterates over the constant periods of that variable. The extent of that loop is the portion of the block in which that variable is active. (Some optimizations can eliminate the WHILE loop, as in the example above: the while loop is implicitly resident in the INSERT statement.) A WHILE statement over a time-varying SQL statement will thus be transformed to two WHILE statements, the outer one over constant periods (of the SQL statement and the time-varying context thus far) and an inner one over the tuples within that particular constant period. On the other hand, if no new time-varying activity is introduced by any portion of the statement, the statement can remain as is. Finally, external routines are mapped as they are in maximal slicing.

VII. Performance Study

How might these two quite different time-slicing techniques perform? Intuitively, there should be queries and data that favor each approach. If a sequenced query specifies a very short valid-time period as its temporal context, maximally-fragmented slicing should perform better because it has the less complex statements in the routine and a few calls to each routine. On the other hand, if a sequenced query requires the result for a very long valid-time period and the data changes frequently in this period, the number of calls to the routine could be large for the maximally-fragmented slicing. In this case, per-statement slicing may outperform maximally-fragmented slicing. We now empirically evaluate the performance.

A. The τPSM Benchmark

To perform our evaluation, we create and use the τPSM benchmark, which is now part of τBench [21]. τBench is a set of temporal and non-temporal benchmarks in the XML and relational formats, created by the authors. τBench is built upon XBench, a family of benchmarks with XML documents, XML Schemas, and associated XQuery queries [22]. One of the benchmarks in XBench, called the *document-centric/single document* (DC/SD) benchmark, defines a book store catalog with a series of books, their authors and publishers, and related books. XBench can randomly

generate the DC/SD benchmark in any of four sizes: small (10MB), normal (100MB), large (1GB), and huge (10GB).

1) Data Sets: τBench provides a family of temporal and non-temporal benchmarks, all based on the original DC/SD XBench benchmark, including the PSM and τPSM benchmarks [21]. For the former, τBench shreds the data into tables. For the latter, τBench begins with a simulation to transform the DC/SD dataset into a temporal dataset. This simulation step involves randomly changing data elements at specific points in time. A set of user-supplied parameters controls the simulation, such as how many elements to change and how often to change them. Then τBench shreds this XML data into the following six temporal tables: `item` (books), `author`, `publisher`, `related_items`, `item_author` (to transform items to authors), and `item_publisher` (to transform items to publishers).

We used three datasets in our experiments: DS1, DS2, and DS3. DS1 contains weekly changes, thus it contains 104 slices over two years, with each item having the same probability of being changed. Each time step experiences a total of 240 changes; thus there are 25K changes in all. DS2 contains the same number of slices but with rows in related tables associated with particular items changed more often (using a Gaussian distribution), to simulate hot-spot items. DS3 returns to the uniform model for the related tuples to be changed, but the changes are carried out on a daily basis, or 693 slices in all, each with 240 changes, or 25K changes in all (the number of slices was chosen to render the same number of total changes). These datasets come in different sizes: SMALL (e.g., DS1.SMALL is 12MB in six tables), MEDIUM (34MB), and LARGE (260MB).

2) Queries: The PSM benchmark contains 16 queries drawn from the 19 queries in XBench (some of the XBench queries were too specific to XML to be transformed to PSM). Each PSM query highlights a feature. Query *q2* highlights the construct of `SET` with a `SELECT` row, *2b* multiple `SET` statements, *q3* a `RETURN` with a `SELECT` row, *q5* a function in the `SELECT` statement, *q6* the `CASE` statement, *q7* the `WHILE` statement, *q7b* the `REPEAT` statement, *q8* a loop name with the `FOR` statement, *q9* a `CALL` within a procedure, *q10* an `IF` without a `CURSOR`, *q11* creation of a temporary table, *q14* a local cursor declaration with associated `FETCH`, `OPEN`, and `CLOSE` statements, *q17* the `LEAVE` statement, *q17b* a non-nested `FETCH` statement, *q19* a function called in the `FROM` clause, and *q20* a `SET` statement. (Some queries, such as *q2*, were also changed to highlight a different feature, such as multiple `SET` statements in *q2b*. See also *q7b* and *q17b*.)

We extended each of these queries by prepending the keyword `VALIDTIME` to render a sequenced variant. We then transformed each according to the maximally-fragmented slicing (abbreviated here as MAX) and per-statement slicing (abbreviated here as PERST) approaches discussed above. Finally, we transformed each of these versions to their equivalent in DB2's syntax. The entire set of queries is available on the fourth author's website [21].

Query *q2* is the SQL query in Figure 3 along with the associated stored function `get_author_name()` given in Figure 1. The MAX version is shown in Figures 9 and 10; the PERST version is provided in Figure 11.

Query *q17b* is notable in that it has a *non-nested* `FETCH` statement. There is an outer loop that includes a fetch from the `all_items_cur` cursor at the very end of the loop. But within the loop is a call to `has_canadian_author`, which returns a temporal result, and a call to `is_small_book`, which also returns a temporal result. Both of these require a `FOR` loop. The effect is that there is a while loop on the original cursor enclosing nested for loops on the temporal results, enclosing code including a fetch of the outer cursor. It is that last piece that cannot be accommodated by the per-statement transformation. Hence, there are no timings for *q17b* for the per-statement transformation in any of the experiments. (We emphasize that MAX always applies, so the entire PSM language is accommodated.)

B. Experiments

We performed a series of experiments to examine the feasibility of utilizing the transformation strategy. We compared the performance between MAX and PERST over a range of several factors: data set (which considers both distribution of changes and number of changes per time step), data set size (small, medium, large), length of the temporal context, and query (which gets at impact of language constructs).

All experiments were conducted on a 2.4GHz Intel Core 2 machine with 2GB of RAM and one 320GB hard drive running Fedora 6 64bit. We chose DB2 (Version 9.1) as the underlying SQL engine, primarily because its PL/SQL supports most of the functions in standard SQL/PSM. However, we had to modify some of the queries to make them acceptable to DB2. For example, queries *q3*, *q11*, and *q14* all use the SQL keyword `BETWEEN`; this predicate needs to be transformed to two less-than-or-equal predicates. The list of about a dozen inconsistences is provided elsewhere [21, Appendix D]. For all the database settings, we used default settings provided by DB2. We performed the experiments with a warm cache to focus on CPU performance.

We started with the sixteen nontemporal PSM queries in τBench. The original queries totalled 500 lines of SQL. We first transformed these queries into the DB2 PSM syntax, adding about forty lines. We then transformed each into its maximal slicing (1600 lines) and per-statement slicing variants (2000 lines). Hence, the nontemporal queries, at about 30 lines each, expanded to 100 lines (maximal) to 125 lines (per-statement). (Recall that all the user had to do was to prepend `VALIDTIME` to the SQL query.)

To ensure that our transformations were correct, we compared the result of evaluating each nontemporal query on a timeslice of the temporal database on each day with the result of a timeslice on that day of the result of both transformations of the temporal version of the query on the temporal database, termed *commutativity* [23]. We also ensured that the results of maximal slicing and per-statement slices were equivalent, and were also equivalent to the union of slices produced

by their nontemporal variant. These tests indicate that the transformations accurately reflected the sequenced semantics.

Query *2* asks for author "Ben." However, the generated data does not contain any records with Ben. To avoid the query returning an empty result set, in which case the DBMS could apply some optimization and thus invalidate meaningful run time measurements, we change the query to look for a valid author that *is* present in the data.

C. Length of Temporal Context

First, we varied the length of the temporal context used in the sequenced query, selected from one day, one week, one month, and one year. (Recall that the data sets contain two years of data.) We performed experiments with both large and small datasets. The results for DS1-SMALL are presented in Figure 12 and for DS1-LARGE in Figure 13, respectively.

In these plots, the x-axis is the length of the temporal context ("d" denotes one day; "y" denotes one year) and the y-axis shows running time in seconds on a logarithmic scale. Two plots are given for each query: MAX with a solid line and circles for points and PERST with a dotted line and triangles for points. So MAX for *q2* for a temporal scope of one day on DS1-SMALL took about $10^{-0.7}$ or 200 milliseconds, whereas that query with a temporal scope of one year ran about $10^{0.8}$ or six seconds. (The actual values in seconds were 0.21, 0.19, 0.67, and 6.6 seconds for MAX and 0.31, 0.29, 0.34, and 0.32 seconds for PERST.)

Examining the trends in Figure 12, using DS1-SMALL, four classes of queries are observed. For class A, per-statement slicing is always faster: queries *q7*, *q7b*, *q11*, *q14*, and *q19*. Class B is more interesting: for queries *q2*, *q2b*, *q3*, *q6*, and *q8*, PERST becomes faster than MAX for a temporal context of between one week and one month. For *q17*, MAX is always faster than PERST; we call this class C. For the remaining queries, comprising class D, MAX starts off faster, but approaches or meets PERST at a long temporal context: *q5*, *q9*, *q10*, and *q20*.

Similar trends can be observed for data set DS1-LARGE (260MB versus 12MB), shown in Figure 13, with some queries about two orders of magnitude slower, which is to be expected. (For *q5*, both perform almost identically.) A few queries move between classes. Queries *q3* and *q6* move from class B to class A. *q9* and *q10* move from class D to class B, all due to MAX getting relatively slower. Interestingly, *q7* and *q7b* change from class A to class C. We provide an explanation of such behavior shortly. With the large dataset, PERST is relatively flat, whereas for most queries MAX has what appears to be a linear component as the temporal context grows from 1 day to 7 days to 30 days to 365 days.

It seems that two effects are in play. We observe a break-even point between the two strategies, which can be explained in that for a very short query time period, the overhead of creating all the constant periods is low and given the simplicity of maximal slicing, the running time of MAX could be faster than a more complicated PERST query.

We also postulate that the execution time of MAX increases significantly because the routine is repeatedly invoked from the `WHERE` clause in the SQL query. The number of times a routine is invoked is determined by the number of satisfying tuples. Therefore, a routine can be called many times. On the other hand, in PERST the routine is called only once. As shown by both figures, running time for PERST is fairly constant.

PERST versions of queries *q7* and *q7b* are quite slow on the large dataset. Interestingly, as shown in Figure 12, these two queries show significant increase in running time even for PERST. The reason is these two queries require cursors to be processed on a per-period basis. Specifically, for each time period, the records need to be processed individually from other periods. Therefore, an auxiliary table is needed to temporarily store the period-based records. As rows are inserted into this auxiliary table for each time period, the transaction log in the DBMS rapidly fills up. Especially when the temporal context is longer, the number of time periods to be processed is higher and writing the logs takes longer. Query *q17* is similar to these two queries. Therefore, the requirement of writing logs significantly impacts the performance of these queries.

In summary, we found that PERST in general outperforms MAX, especially with a long temporal context and for larger data sets, probably because PERST only invokes a routine once while MAX invokes a routine many times. For certain types of queries, MAX is required.

D. Scalability

While Figure 13 somewhat gets at scalability, the plots in Figure 14 do so directly. Here the x-axis is dataset size: 'S' denotes SMALL, 'M', MEDIUM, and 'L', LARGE. For most of the queries, we observe that as the dataset size increases, the running time of a query also increases. There are two exceptions, *q7b* and *q17b*, showing a decrease in running time from small to medium, for the maximal slicing approach. Due to the size difference of the datasets, a different query plan can be used for each of these datasets, even with an identical SQL query statement. MAX requires a routine to be executed many times, which accentuates the performance difference. In particular, if the plans for executing the SQL statements are even a little slower in the small dataset than in the medium, a significant performance difference can be expected.

E. Varying Number of Slices and Data Distribution

Figure 15 shows a comparison among the three datasets (the SMALL version of each). Increasing the number of slices (compare DS1 with DS3) appears to have a significant impact on performance, especially for MAX. Increasing the skew of the data (from uniform to Gaussian distribution: DS1 compared with DS2) produces a decrease in running time for maximal slicing on queries *q2* and *q2b*. For these two queries, one of the predicates asks for a particular item that is not a hot spot. There are thus fewer changes to this selected item, and processing a fewer number of records results in a shorter running time. (Presumably if the selected data was in a hot spot, exactly the opposite would occur.)

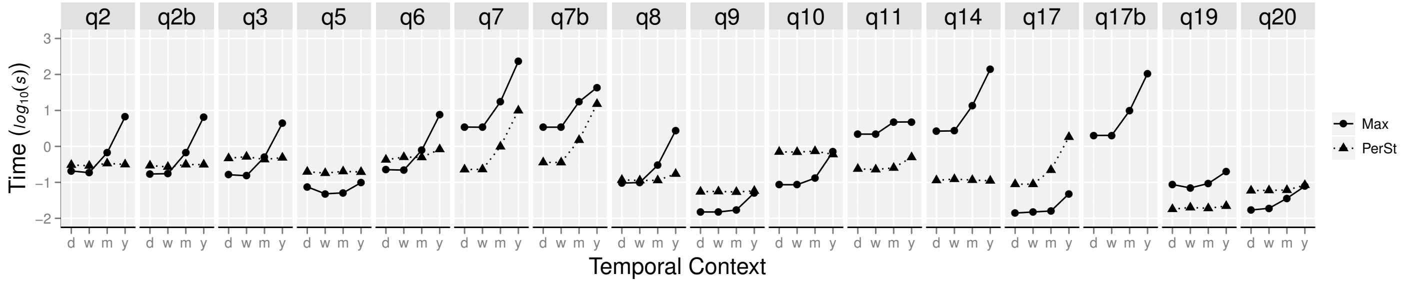

Fig. 12. Varying Time Period on the Small Dataset (DS1-SMALL).

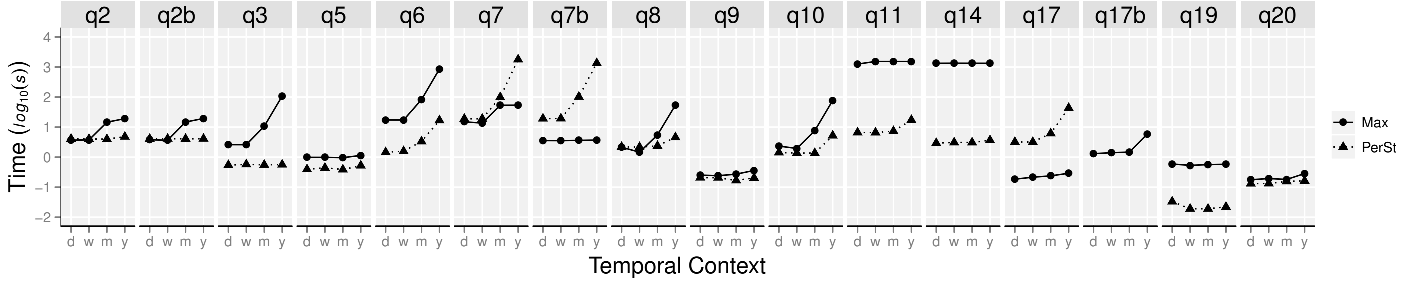

Fig. 13. Varying Time Period on the Large Dataset (DS1-LARGE).

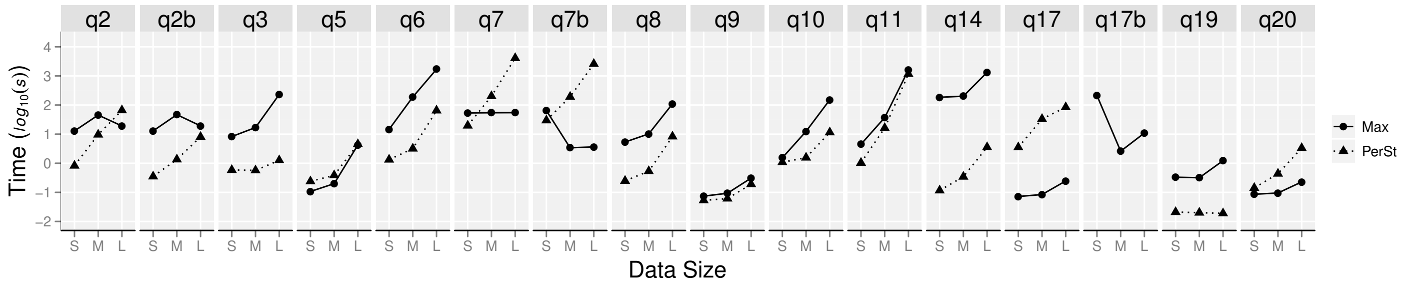

Fig. 14. Varying Data Set Size (for Data Set DS1, S = 12MB; M = 34MB; L = 260MB).

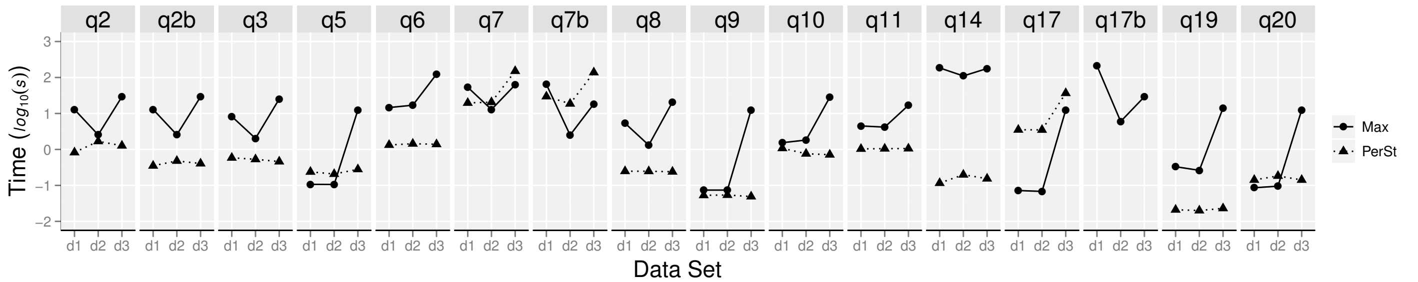

Fig. 15. Varying Data Characteristics (DS1-SMALL (Weekly, Uniform), DS2-SMALL (Weekly, Gaussian), and DS3-SMALL (Daily, Uniform)).

In general, DS1 and DS2 show no significant difference, whereas DS3 causes longer running times. We conclude that the number of slices is a dominant factor in determining the performance of queries, especially for MAX.

F. Choosing the Approach

Of the 160 data points reported in Figures 12–15, PERST is faster for about 70% of these. It appears that a query optimizer should choose that slicing method unless (a) the transformation rules don't work for PERST, e.g., non-nested FETCHes as discussed at the end of Section VII-A2 (15 data points), (b) cursors are required on a per-period basis by PERST *and* the data set is large, which as the case for queries *q7* and *q7b* (22 out of 28 data points), or (c) the query is on a small database *and* has a short temporal context (27 out of 41 data points). This multi-faceted heuristic works most of the time, though it selects the wrong slicing method in about 13% of the cases. Note though that some of the per-statement functions are very complex and are being evaluated by a sophisticated DBMS, making general statements about relative performance difficult.

VIII. Summary and Future Work

PSM is now mature technology, and several DBMS vendors now include substantial support for temporal queries. How should PSM be extended so that temporal upward compatibility is ensured and that the full functionality of PSM can be applied to tables with valid-time and transaction-time support? Our proposal is to apply the notion of temporal modifier, already in play in SQL/Temporal, full-force to PSM. How can Temporal SQL/PSM be implemented in an efficient manner? Does the stratum approach even work in this case? What optimizations can be applied to render a more efficient implementation? The maximally-fragmented slicing transformation is relatively simple. In fact, the routine is largely unaffected, other than being passed a date to be used in queries contained in it, and passed on to other routines. The bulk of the work is done before the query itself is executed

The maximally-fragmented transformation always applies. However, because it possibly renders many tiny snippets and invokes the routine (and thus, subsequently-invoked other routines) on a per constant basis, such a transformation might be slow. Thus we developed per-statement slicing, which slices at the statement level, involving fewer routine calls but more complex routines. The per-statement mapping is not complete.

Our empirical study showed that the length of the temporal context has a significant impact on the performance of MAX: the shorter the query period, the fewer the slices, therefore, the better the performance of MAX. This simple transformation could be better than PERST when the query period is short enough. However, when the query period is longer, the performance of MAX drops dramatically.

This paper offers for the first time a well-defined syntax and semantics and at least two possible implementation strategies, providing a way forward for vendors to extend their support of temporal queries to include invocation of routines.

There are several avenues for further investigation. The transformations should be automated, so that users don't need to see the complexities, and refined to work with the Oracle and Teradata-specific variants of PSM. Also to be worked out is support for bitemporal data. We are also considering further optimizations to the transformation. While we have empirical evidence that our transformations are correct, it would be helpful to use a formal semantics of SQL and of PSM to prove their correctness. It would also be useful to develop a cost model that can predict which transformation will perform better, to replace the heuristic in Section VII-F. We note that the τBench queries form a *micro-benchmark*, in that each query generally focuses on one construct. A more detailed analysis of the performance figures for these queries might shed light on how a complex routine containing multiple constructs might perform. Finally, maximal and per-statement slicing provide endpoints of a spectrum. Each statement provides a possible location for slicing, with each location slicing on one or more underlying temporal tables. Ideally the transformation would decide exactly where and how aggressive each slicing should be to maximize performance.

IX. Acknowledgments

This research was supported in part by NSF grants IIS-0415101 and IIS-0803229 and a grant from Microsoft.

References

[1] L. Liu and M. Özsu, Eds., *Encyclopedia of Database Systems*. Springer, 2009.

[2] J. Melton, "Understanding SQL's Stored Procedures: A Complete Guide to SQL/PSM," Morgan Kaufmann Publishers, 1998.

[3] ISO/IEC, "Information Technology—Database Languages—SQL 9075," 2003.

[4] R. T. Snodgrass, "Developing Time-Oriented Database Applications in SQL," Morgan Kaufmann Publishers, 2002.

[5] G. Özsoyoglu and R. T. Snodgrass, "Temporal and Real-time Databases: A Survey," *IEEE Transactions on Knowledge and Data Engineering*, vol. 7, no. 4, pp. 513–532, August 1995.

[6] R. T. Snodgrass (editor), I. Ahn, G. Ariav, D. S. Batory, J. Clifford, C. E. Dyreson, R. Elmasri, F. Grandi, C. S. Jensen, W. Kafer, N. Kline, K. Kulkanri, T. Y. C. Leung, N. Lorentzos, J. F. Roddick, A. Segev, M. D. Soo, and S. M. Sripada, "The Temporal Query Language TSQL2," Kluwer Academic Pub, 1995.

[7] A. Tansel, J. Clifford, S. Jajodia, A. Segev, and R. T. S. (eds.), "Temporal Databases: Theory, Design, and Implementation. Database Systems and Applications Series," Benjamin/Cummings, Redwood City, CA, 1993.

[8] R. T. Snodgrass, M. H. Böhlen, C. S. Jensen, and A. Steiner, "Adding Transaction Time to SQL/Temporal," SQL3 Standards Committee, Change proposal ANSI X3H2-96-502r2, ISO/IEC JTC1/SC21/ WG3 DBL MAD-147r2, Nov. 1996.

[9] ——, "Adding Valid Time to SQL/Temporal," SQL3 Standards Committee, Change proposal ANSI X3H2-96-501r2, ISO/IEC JTC1/SC21/ WG3 DBL MAD-146r2, Nov. 1996.

[10] Oracle Corp., "Workspace Manager Developer's Guide 11g Release 1 (11.1)," Aug. 2008.

[11] Teradata Corp., "Teradata Temporal," Oct. 2010, http://www.teradata.com/t/database/Teradata-Temporal/, accessed October 2010.

[12] IBM Corp., "A Matter of Time: Temporal Data Management in DB2 for z/OS," Dec. 2010, accessed December 2010. [Online]. Available: https://www14.software.ibm.com/webapp/iwm/web/signup.do?lang=en_US&source=sw-infomgt&S_PKG=db2z-temporal-tables-wp

[13] R. T. Snodgrass and I. Ahn, "Temporal Databases," *IEEE Computer*, vol. 19, no. 9, pp. 35–42, September 1986.

[14] J. Bair, M. Böhlen, C. S. Jensen, and R. T. Snodgrass, "Notions of Upward Compatibility of Temporal Query Languages," *Business Informatics (Wirtschafts Informatik)*, vol. 39, no. 1, pp. 25–34, February 1997.

[15] M. H. Böhlen, C. S. Jensen, and R. T. Snodgrass, "Temporal Statement Modifiers," *ACM Transactions on Database Systems*, vol. 25, no. 4, pp. 407–456, 2000.

[16] K. Torp, C. S. Jensen, and R. T. Snodgrass, "Stratum Approachs to Temporal DBMS Implementation," in *Proceedings of IDEAS*, Cardiff, Wales, UK, 1998, pp. 4–13.

[17] D. Gao, "Supporting the procedural component of query languages over time-varying data," Ph.D. dissertation, Computer Science Department, University of Arizona, Apr. 2009.

[18] J. E. Stoy, "Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory," The MIT Press, 1979.

[19] D. Gao and R. T. Snodgrass, "Temporal Slicing in the Evaluation of XML Queries," in *Proceedings of International Conference on Very Large Data Bases*, Berlin, Germany, September 2003, pp. 632–643.

[20] R. T. Snodgrass, S. Gomez, and L. E. McKenzie, "Aggregates in the Temporal Query Language TQuel," *IEEE Transactions on Knowledge and Data Engineering*, vol. 5, no. 5, pp. 826–842, 1993.

[21] S. W. Thomas, R. T. Snodgrass, and R. Zhang, "τBench: Extending XBench with Time," *TimeCenter*, 2010, TR-92, accessed December 2010. [Online]. Available: http://cs.queensu.ca/~sthomas

[22] B. B. Yao, M. T. Özsu, and N. Khandelwal, "Xbench benchmark and performance testing of XML DBMSs," in *Proceedings of ICDE*, 2004, pp. 621–633.

[23] D. Gao, C. S. Jensen, R. T. Snodgrass, and M. Soo, "Join Operations in Temporal Databases," *International Journal on Very Large Databases*, vol. 14, p. 229, March 2005.