Layered Implementation of Temporal DBMSs—Concepts and Techniques

Kristian Torp Christian S. Jensen Michael Böhlen

TR-2

A TimeCenter Technical Report

Title	**Layered Implementation of Temporal DBMSs—Concepts and Techniques**
Author(s)	Kristian Torp Christian S. Jensen Michael Böhlen
Publication History	A shorter version of this paper appeared in DASFAA 97

TIMECENTER Participants

Aalborg University, Denmark
Michael H. Böhlen
Renato Busatto
Heidi Gregersen
Christian S. Jensen (codirector)
Kristian Torp

University of Arizona, USA
Anindya Datta
Hong Lin
Richard T. Snodgrass (codirector)

Individual participants
Curtis E. Dyreson, James Cook University, Australia
Michael D. Soo, University of South Florida, USA
Andreas Steiner, ETH Zurich, Switzerland

The TIMECENTER icon on the cover combines two "arrows." These "arrows" are letters in the so-called *Rune* alphabet used one millennium ago by the Vikings, as well as by their precedessors and successors, The Rune alphabet (second phase) has 16 letters. They all have angular shapes and lack horizontal lines because the primary storage medium was wood. However, runes may also be found on jewelry, tools, and weapons. Runes were perceived by many as having magic, hidden powers.

The two Rune arrows in the icon denote "T" and "C," respectively.

Abstract

A wide range of database applications manage time-varying data. Examples include, e.g., accounting, personnel, schedule, and data warehousing applications. At the same time, it is well-known that querying and correctly updating time-varying data is difficult and error-prone when using standard SQL. As a result of a decade of intensive exploration, temporal extensions of SQL have reached a level of maturity and sophistication where it is clear that they offer substantial benefits over SQL when managing time-varying data.

The topic of this paper is the effective implementation of temporally extended SQL's. Traditionally, it has been assumed that a temporal DBMS must be built from scratch, utilizing new technologies for storage, indexing, query optimization, concurrency control, and recovery. This paper adopts a quite different approach. Specifically, it explores the concepts and techniques involved in implementing a temporally enhanced SQL while maximally reusing the facilities of an existing SQL implementation, e.g., Oracle or DB2. The topics covered span the choice of an adequate timestamp domain that include the time variable "now," a comparison of alternative query processing architectures including a partial parser approach, update processing, and transaction processing, the latter including how to ensure ACID properties and assign correct timestamps.

Keywords: Temporal databases, SQL, layered architecture, query processing

1 Introduction

A wide variety of existing database applications manage time-varying data (e.g., see [12, p. 670] [21]). Examples include medical, banking, insurance, and data warehousing applications.

At the same time, it is widely recognized that temporal data management in SQL–92 is a complicated and error-prone proposition. Updates and queries on temporal data are complex and are thus hard to formulate correctly and subsequently understand (e.g., see [16, 10, 26]). This insight is also not new, and following more than a decade of research, advanced query languages with built-in temporal support now exist (e.g., [25, 33]) that substantially simplify temporal data management.

To be applicable in practice, a temporal language must meet the challenges of legacy code. Specifically, a temporal query language should be upward compatible with SQL–92, meaning that the operation of the bulks of legacy code is not affected when temporal support is adopted. In addition, it is desirable that the languages permit for incremental exploitation of the temporal support. When a temporal language is first adopted, no existing application code takes advantage of the temporal features. Only when converting old applications and developing new ones are the benefits of temporal support achieved. To be able to make this transition to temporal support, temporal and old, "non-temporal" applications must be able to coexist smoothly.

Temporal query languages effectively move complexity from the user's application to the implementation of the DBMS. The usual architecture adopted when building a temporal DBMS is the *integrated architecture* used for implementing commercial relational DBMS's [1, 2, 30, 29, 33]. This architecture allows the implementor maximum flexibility in implementing the temporal query language. This flexibility may potentially be used for developing an efficient implementation that makes use of, e.g., special-purpose indices, query optimizers, storage structures, and transaction management techniques. However, developing a temporal DBMS with this approach is also very time consuming and resource intensive. Indeed, existing implementations of temporal query languages using this architecture focus on proof of concept or functionality only—they do not also succeed in exploiting the potential for improved performance [6]. A main reason why the layered architecture has received only little attention so far is that the ambitious performance goal has been to achieve the same performance in a temporal database (with multiple versions of data) as in a snapshot database (without versions and thus with much less data).

As a supplement, we will in this paper explore the implementation of temporal query languages using a different architecture, the *layered architecture*. This architecture implements a temporal query language on top of an existing relational DBMS. Here, the relational DBMS is considered a black box, in that it is not possible to modify its implementation when building the temporal DBMS.

The most important potential advantage of building on top of an existing DBMS, compared to building the temporal DBMS from scratch, is the possibility of reusing the services of the DBMS, e.g., the concurrency control and recovery mechanisms, for implementing the extended functionality. Another

advantage is the possibility of achieving upward compatibility with a minimal coding effort. The major disadvantages are the entry costs that DBMS'es impose on its clients, as well as the impossibility of directly manipulating DBMS-internal data structures.

In this paper we focus on the design of a temporal DBMS, implemented using a layered architecture. Our main design goals are upward compatibility and maximum reuse of the underlying DBMS. Another goal is that no queries should experience significantly lower performance when replacing an existing DBMS with a temporal DBMS. Throughout, we aim to achieve these goals.

We consider the possible alternatives for a domain for timestamps, including the possible values available for representing the time variable "now." We show how a *partial parser architecture* can be used for achieving upward compatibility with a minimal effort. Several implementation aspects are covered in details. Specifically, we show how correct update processing is achieved, and we discuss the possibilities for implementing correct temporal-transaction processing.

A partial parser has been implemented that provides a minimum of temporal support, and all code examples in the paper have been tested using the Oracle DBMS and its query language, PL-SQL. We have chosen a commercial DBMS as the underlying DBMS, and not an extensible system [13, 31, 7, 3] because we want to investigate the seamless migration of legacy systems.

Related research may be divided into two parts: other work on the layered implementation of temporal query languages and work that uses a layered architecture for implementing other functionality.

The research reported by Vassilakis et al. [34] assumes a layered implementation of an interval-extended query language, VT-SQL, on top of Ingres. The focus is on correct transaction support, and the problem addressed is that the integrity of transactions may be violated when the layer uses temporary tables to store intermediate results. This is because the SQL–92 standard does not require that a DBMS permits both data manipulation and data definition statements to be executed in the same transaction [17, p. 76]. This may make it impossible to rollback a transaction started from the layer. The problem is solved by using two connections to the DBMS. We find that the problem is eliminated if the DBMS supports SQL–92 temporary tables, and this paper does not address that problem.

The TSQL2 query language was designed with ease of implementation in mind, but still the idea was to exploit well-known temporal implementation techniques and available internal modules (e.g., for storage management) when building an integrated implementation [27]. Layered implementation was not considered.

More recently, a layered implementation, based on ORACLE, was pursued in the TimeDB prototype [28] that implements the ATSQL query language, which incorporates ideas from TSQL2 [27] and ChronoLog [4]. The topics covered in this paper are partly inspired by and generalize TimeDB.

The layered architecture has been used for implementing object-oriented data models using the services of relational DBMS's [32, 15]. Here, the layer is used to make a paradigm shift from a relational to an object-oriented model, and upward compatibility is not considered. In other research, a passive DBMS (Starburst) was converted to an active DBMS [20]. A small extension to SQL was implemented using a layered architecture, with the goal of maximum reuse of the underlying, passive DBMS.

The rest of the paper is organized as follows. In Section 2 we first characterize temporal support and motivate the need for a temporal SQL. Then the layered architecture is introduced, and the design goals for a layered implementation are given. Section 3 is concerned with the domain of timestamps, including the representation of the timestamp value "now." Alternative partial parser architectures for temporal query processing are the topic of Section 4, and Section 5 addresses the implementation of temporal modification statements. Section 6 is devoted to the correct processing of temporal transactions. Finally, Section 7 summarizes and points to directions of future research.

2 Temporal SQL and Design Goals

Following an introduction to and a motivation for the functionality that a temporal SQL adds to SQL–92, we introduce the layered architecture and state the design goals for the layered implementation of this functionality.

2.1 Temporal Functionality

Two general temporal aspects of database facts have received particular interest [24]. The *transaction time* of a fact records when the fact is current in the database, and is handled by the temporal DBMS. Orthogonally, the *valid time* of a fact records when the fact is true in the modeled reality, and is handled by the user; or default values are supplied by the temporal DBMS. A data model or DBMS with built-in support for both times is called *bitemporal* [14]; and if neither time is supported, it is termed *non-temporal*.

In a temporal data model, a database may record the temporal aspects of data implicitly (because the data model is temporal) or explicitly (because explicit attributes are available), or both. To distinguish between these two, we adopt the terminology outlined below.

$$\textbf{Implicit Time Support} \left\{ \begin{array}{c} \text{“snapshot”} \\ \text{“valid-time”} \\ \text{“transaction-time”} \\ \text{“bitemporal”} \end{array} \right\} \times \text{“relation with”} \times \textbf{Explicit Time Support} \left\{ \begin{array}{c} \text{“snapshot”} \\ \text{“valid-time”} \\ \text{“transaction-time”} \\ \text{“bitemporal”} \end{array} \right\} \text{“data”}$$

For example, the relation `Employee` in Figure 1 is a bitemporal relation with snapshot data. The relation records department information for employees, with a granularity of days, and with the last four implicit columns encoding the transaction-time and valid-time dimensions using half-open intervals.

Name	Department	T-Start	T-Stop	V-Begin	V-End
Torben	Toy	8-8-1996	19-8-1996	10-8-1996	*NOW*
Alex	Sports	12-8-1996	*NOW*	23-8-1996	31-8-1996
Torben	Toy	19-8-1996	*NOW*	10-8-1996	21-8-1996
Torben	Sports	19-8-1996	*NOW*	21-8-1996	*NOW*

Figure 1: The Bitemporal Relation, `Employee`

On August 8, the tuple (Torben, Toy, 8-8-1996, *NOW*, 10-8-1996, *NOW*) was inserted, meaning that Torben will be in the Toy department from August 10 until the current time. Similarly, at August 12 we recorded that Alex was to be in the Sports department from August 23 and until August 31. Later, on August 19, we learned that Torben was to start in the Sports department on August 21. The relation was subsequently updated to record the new belief. We no longer believed that Torben would be in the Toy department from August 10 until the current time, but believed instead that he would be there only until August 21 and that he would be in Sports from August 21 and until the current time. Thus, the first *NOW* in the original tuple is changed to August 19, and the resulting two new tuples with our new beliefs are inserted.

The relation in Figure 1 shows that the data model for a temporal SQL is different from the SQL–92 data model. Four implicit attributes have been added to the data structure (the relation).

Instead of assuming a specific temporal query language, we will simply assume that the temporal query language supports standard temporal database functionality, as it may be found in the various existing data models. We also assume that the temporal data model satisfies three important properties, namely upward compatibility (UC), temporal upward compatibility (TUC) [5] and snapshot reducibility (SR) [22]. We will define these properties next.

There are two requirements for a new data model to be *upward compatible* with respect to an old data model [5]. First, the data structures of the new data model must be a superset of the data structures in the old data model. Second, a legal statement in the old data model must also be a legal statement in the new data model, and the semantics must be the same, e.g., the result returned by a query must be the same. A temporal extension will invariably include new key words. We assume that such key words do not occur in legacy statements as identifiers (e.g., as table names). Legacy code must be inspected to avoid such occurrences. For a temporal data model and DBMS to be successful, it is important that these requirements are fulfilled with respect to SQL–92.

TUC is a more restrictive requirement. For a new temporal data model to be *temporal upward compatible* with respect to an old data model, it is required that all legacy statements work unchanged even when the tables they use are changed to provide built-in support for transaction time or valid time.

Thus TUC poses special requirements on the effect of legacy modification statements applied to temporal relations and to the processing of legacy queries on such relations.

When a temporal DBMS is taken in use, the application code does not immediately exploit the added functionality; rather, the temporal features are only realized incrementally, as new applications are developed and legacy application are being modernized. TUC guarantees that legacy and new applications may coexist harmoniously.

Lastly, a temporal statement is snapshot reducible with respect to a non-temporal statement if all snapshots of the result of the temporal statement are the same as the result of the non-temporal statement evaluated on the corresponding snapshots of the argument relations. The idea of SR is that the expertise gained by application programmers using SQL–92 should be applicable to the added temporal functionality, making it easier to understand and use the new facilities [23].

Using TUC and SR, we may divide the new statements in a temporal SQL into three categories. First, *TUC statements* are conventional SQL–92 statements, with the exception that they involve temporal relations. These statements are not "aware" of the temporal extensions and access only the current state of the temporal database. Second, *sequenced statements* are those statements that are defined by means of snapshot reducibility to a corresponding SQL–92 query. Third, *non-sequenced statements* are statements that do not have a corresponding SQL–92 counterpart. These statements exploit the temporal facilities, but do not rely on the DBMS to do timestamp-related processing according to snapshot reducibility. Rather, they specify non-default timestamp manipulation.

2.2 The Layered Architecture

The layered architecture implements new temporal query facilities in SQL–92. The queries written in the new temporal language are then converted to SQL–92 queries that are subsequently executed by the underlying DBMS. No conversion is needed for plain SQL–92 queries. The layered temporal database architecture is shown in Figure 2.

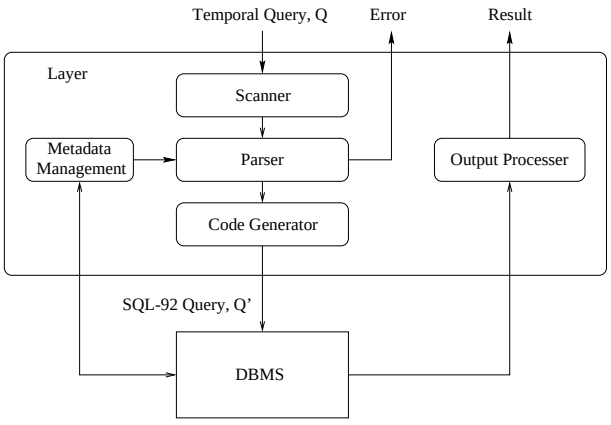

Figure 2: The Layered Temporal Database Architecture

Some comments are in order. First, the layer uses the DBMS as a "black box." Second, the assumed control structures in this architecture are simple. The layer converts temporal queries to SQL–92 queries, keeps track of information used by the layer internally, and does some post-processing of the result received from the DBMS. More precisely, a transaction with temporal statements is compiled into a single SQL–92 transaction that is executed on the DBMS without interference from the layer—the layer simply receives the result and applies some post-processing. There is thus no control module in the layer, and there is minimal interaction between the layer and the DBMS. While maximizing the independence among the two components, this simplicity also restricts the options available for implementing temporal queries in the layer. The specific impacts on the functionality of the temporal query language and on performance are not yet well understood.

As a simple example of how the layer converts a temporal query into an SQL–92 query, consider the sequenced temporal query in Figure 3A that finds the name and department of employees in the sports department and how long they have been there. The new keywords `SEQUENCED VALID` indicate that the query should be computed over all (valid) times, but just for the current (transaction time) state. The query is converted to the SQL–92 query shown in Figure 3B (we will elaborate on this translation later in the paper). If evaluated on August 25, the query returns the relation in Figure 4.

```
SEQUENCED VALID
  SELECT Name, Department
  FROM Employee
  WHERE Department = 'Sports'
```

A

```
SELECT Name, Department, V-Begin, V-End
FROM Employee
WHERE Department = 'Sports' AND
      T-Start <= CURRENT_TIME AND
      T-Stop  >= CURRENT_TIME AND
```

B

Figure 3: Conversion of a Temporal Query to an SQL–92 Query

Name	Department	V-Begin	V-End
Torben	Sports	21-8-1996	*NOW*
Alex	Sports	23-8-1996	31-8-1996

Figure 4: The Result of the Query in Figure 3, at August 25 1996

The query in Figure 3 is written in the temporal query language ATSQL [28]. It is beyond the scope of this paper to define syntax and semantics of this language. However, the extensions are consistent with SQL–92 and are easy to understand. In the example above the `SEQUENCED VALID` keywords indicate a sequenced (and thus snapshot reduc ible) semantics as discussed in the previous section. We will use ATSQL as an example of a temporal SQL throughout this paper.

2.3 Design Goals

In implementing the layered temporal DBMS, we stress seven somewhat conflicting and overlapping design goals, namely achieving upward compatibility with a minimal coding effort, gradual availability of temporal functionality, achieving temporal upward compatibility, maximum reuse of existing relational database technology, retention of all desired properties of the underlying DBMS, platform independence, and adequate performance. We discuss each in turn.

As discussed already, UC is important in order to be able to protect the investments in legacy code. Achieving UC with a minimal effort and gradual availability of advanced functionality are related goals. First, it should be possible to exploit in the layered architecture that the underlying DBMS already supports SQL–92. Second, it should be possible to make the new temporal functionality available stepwise. Satisfying these goals provides a foundation for early availability of a succession of working temporal DBMSs with increasing functionality.

TUC makes it possible to turn an existing snapshot database into a temporal database, without affecting legacy code. The old applications work exactly as in the legacy DBMS, and new applications can take advantages of the temporal functionality added to the database. TUC helps achieve a smooth, evolutionary integration of temporal support into an organization.

Few software companies have the resources for building a temporal DBMS from scratch. By aiming for maximum reuse of existing technology, we are striving towards a feasible implementation where both SQL–92 and temporal queries are processed by the underlying DBMS. Only temporal features not found in the DBMS are implemented in the layer.

It is important to retain all the desirable properties of the underlying DBMS. For example, we want to retain ACID properties. With this goal we want to assure that we are adding to the underlying DBMS. However, this also means that if the underlying DBMS does not have a certain core database property, the temporal DBMS will not have it either.

We stress platform independence because we want the layer to be independent of the underlying DBMS. By generating SQL–92 code, the layer should be portable to any DBMS supporting this language.

Rather than attempting to achieve higher performance than existing DBMS's, we simply aim at achieving adequate performance. Specifically, legacy code should be processed with the same speed as in the DBMS, and temporal queries on a temporal database with snapshot data should be as fast as the corresponding SQL–92 queries on the corresponding (i.e., with the same information contents) snapshot database with temporal data.

Achieving all the design goals simultaneously is not always possible. For example, the maximum-reuse goal implies that the layer should be as thin as possible, which is likely to be in conflict with the adequate-performance goal. Similarly, the platform-independence goal may be in conflict with the maximal-reuse goal.

2.4 Fundamental Limitations of an SQL-based Temporal Preprocessor

An important question when adopting a layered architecture is whether it is possible and practical to translate all temporal SQL queries to SQL–92 queries. For most temporal SQL queries, this translation is unproblematic, but not for all, as we shall see next.

There is a scope problem when mapping some temporal queries to SQL–92 queries. Consider the query in Figure 5A. It select the name and valid time of employees that are listed in two different bitemporal employee relations. If the content of `Employee` is (Joe, Shoe, [1981, 1985)), leaving out transaction time, and the content of `Employee1` is (Joe, Sports, [1981, 1983)) then the result of the query is (Joe, [1981, 1983)).

```
A         SEQUENCED VALID
            SELECT E.Name
            FROM  Employee E
            WHERE Name IN (SELECT E1.Name
                           FROM   Employee1 E1)

B  SELECT E.Name
          CASE
             WHEN E.V-Start >= E1.V-Start THEN E1.V-Start
             WHEN E.V-Start <  E1.V-Start THEN E.V-Start
             WHEN E.V-End   >= E1.V-Start THEN E1.V-End
             WHEN E.V-End   <  E1.V-Start THEN E.V-Start
             ELSE 'Error in CASE'
          END
   FROM    Employee E
   WHERE   E.Name = IN (
                SELECT E1. Name
                FROM   Employee1 E1
                WHERE  ((E.V-Start  <= E1.V-Start AND E.V-Start  <  E1.V-End)  OR
                        (E1.V-Start <= E.V-End    AND E.V-End    <  E1.V-End)  OR
                        (E1.V-Start <= E.V-Start  AND E.V-End    <= E1.V-End)  OR
                        (E.V-Start  <= E1.V-Start AND E1.V-End   <= E.V-End)))
```

Figure 5: A Scope Problem

Figure 5B gives the straightforward mapping of the temporal query to SQL–92. Unfortunately, the resulting SQL–92 query is faulty. The correlation name `E1`, which is used to calculate the timestamps of result tuples, is used outside of its scope. At first sight, it may seem that rewriting the query into a join is a solution. But while this transformation does eliminate the scope problem, it introduces new problems. When duplicates are permitted, unnesting is generally not easily possible.

In conclusion, while we believe that much of the functionality of a temporally enhanced SQL may be mapped systematically to SQL–92, there exist temporal queries, e.g., complex nested queries, for which we are now aware of a systematic mapping.

3 Representing the Time Domain

As illustrated in Figure 1, four extra attributes, termed timestamp attributes, and several value-equivalent tuples, i.e., tuples with mutually identical non-timestamp attribute values, are used when recording the temporal aspects of a single fact. The timestamp attributes encode rectangular regions in the space spanned by transaction time and valid time. When a fact has an associated temporal shape composed of several such regions, several tuples are used for representing it.

In this section we will discuss which domain to use for the timestamp attributes and how to represent the special temporal database value "now."

3.1 Choosing the Time Domain

The domain of the timestamped attributes can be one of the SQL–92 datetime data types (`DATE` or `TIMESTAMP`). The advantage of using one of the built-in types is maximum reuse. The disadvantage is that the domain is limited to represent the years 0001 to 9999 [17].

If the limits of the SQL–92 data types is a problem to the applications, the domain of the time attributes can be represented using a new temporal data type handled by the layer, and stored as a `BIT(x)` in the DBMS. The advantage of using a new temporal data type is that it can represent the entire age of the universe [11]. The most obvious disadvantage is that the layer will be thicker, because all handling of the new data type must be implemented in the layer. Further, because dates are irregular, i.e., there are different numbers of days in different months, and because the arithmetic operators defined on the `BIT(x)` data type are regular, we cannot easily use the `BIT(x)` arithmetic operators in the DBMS to manipulate the new data type [9].

As an example of the problems with the irregular arithmetic operators, assume that the finest granularity is days and that we want to evaluate the expression `V-Begin + '1 Month'`. This cannot be done without first decoding `V-Begin` in the layer before adding the correct number of days, e.g., if `V-Begin` is in March, we will have to add 31 days, and if `V-Begin` is in April, we will add 30 days. As a result, the manipulation of time attributes to a large extent must be handled by the layer. This means more tuples have to be sent from the DBMS to the layer to be processed. This again leads to a performance penalty.

Because using `BIT(x)` as the time domain for the time attributes both makes the layer thicker and leads to a performance decrease, we choose to use the built-in data type `TIMESTAMP`. We have here reached one of the limitation on building on top of an existing DBMS: Adding a new data type in the layer, when the underlying DBMS does not support abstract data types, is a major modification.

3.2 Representing "Now" in the Time Domain

Temporal relations may record facts that are valid from or until the current time, and the information they record is or is not current. The relation in Figure 1 exemplifies this representation of "now"-relative information.

The value "now" is not part of the domain of SQL–92 `TIMESTAMP` values, making it necessary to represent "now" by some other value in the domain. A requirement to a useful value is that it is not also used with some other meaning. Otherwise, the meaning of the value becomes overloaded. There are essentially two choices of a value for denoting "now": It is possible to use the value `NULL` or to use a well-chosen "normal" value, e.g., the smallest or largest timestamp. After a general discussion of this approach, we compare the two possibilities.

No matter what value is chosen, this will limit the domain of the data type and create a potential for overloading. For transaction-time attributes, this is not a problem because their values are system supplied. However, for valid-time attributes, this is a real restriction. Furthermore, we have to explicitly treat the value representing "now" specially, e.g., make sure the user does not enter the special value; and when we display data to the user, we have to convert the value used for "now" to an appropriate value (e.g., the string "NOW").

Next, we compare `NULL` with "regular" timestamp values. The value `NULL` has special properties that makes it different from any other value. An advantage of `NULL` is that it takes up less space than a regular timestamp value. Also, the value `NULL` can be processed faster. This aspect is discussed empirically in the next section. (While these observations pertain to Oracle [18], similar statements should hold for

other DBMS's.) A disadvantage of `NULL` is that columns that permit `NULL` values prevent the DBMS from using indices. However, using a non-`NULL` value also impacts indexing adversely. For example, assume that a B^+-tree index, e.g., on `V-End`, is used to retrieve tuples with a time period that overlaps "now." Because "now" is represented by a large or a small value, tuples with the `V-End` attribute set to "now" will not be in the range retrieved. They will thus have to be found at one of the "sides" of the B^+-tree.

3.3 Using "Now" in Queries

Above, we considered the representation of "now" in temporal relations. The next step is to consider the querying of such relations. Here, it is quite easy to contend with the use of either of `NULL`, the minimum value, and the maximum value for "now." Assuming a temporally enhanced SQL, "now" will be used in the `SELECT` and `WHERE` clauses. The idea is to check values of the timestamp attributes and replace them with the current time (i.e., the time when the query is executed) if they are equal to the time representing "now."

For example, in a `SELECT` or `WHERE` clause the valid-time end of tuples in a relation can be referenced as `END(VALID(relation-name))` in ATSQL. This is translated to the following in an SQL–92 query: [1]

```
CASE
     WHEN relation-name.V-End = <value representing now>
     THEN CURRENT_TIMESTAMP
     ELSE relation-name.V-End
END
```

The expression returns the current time if `V-End` is equal to the value representing "now"; otherwise, it returns `V-End`.

3.4 Performance Comparison of Alternative "now" Representations

We have seen that it is possible to use `NULL`, the minimum, and maximum values as representatives for "now." To set the alternatives apart, we compare their performance.

Specifically, for each choice for NOW, we perform each of three different representative timeslice queries on three different relations. We choose timeslice queries because of their importance in temporal query languages [33]. The queries favor the current state, because this state is assumed to be accessed much more frequently than old states. The tests were performed on a SUN Sparc 10 using the Oracle RDBMS version 7.2.2.4. The types of timeslices used are illustrated in Figure 6.

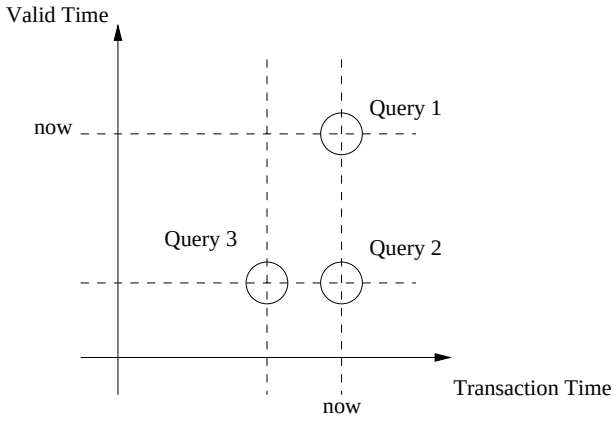

Figure 6: The Three Types of Timeslice Queries in the Test

Query 1 retrieves the current state in both transaction time and valid time, i.e., it selects tuples with transaction-time and valid-time intervals that both overlap with the current time. Tuples with intervals

[1] For using NULL there is a shorthand `COALESCE(relation-name.V-End, CURRENT_TIMESTAMP)`.

that end at "now" thus qualify. Put differently, the query retrieves what we currently believe about the current state of the modeled reality. Query 2 timeslices the argument relation as of "now" in transaction time and as of a past time in valid time. It thus retrieves our current belief about a past state of reality. Query 3 timeslices the relation as of a past time in both transaction time and valid timeand thus retrieves a past belief about a past state of reality. The actual queries may be found in Appendix A.

The queries are performed on three different bitemporal tables, with varying distribution of their tuples. In the first relation, 10% of the tuples overlaps with the current time in both transaction and valid time. In the second and third relations, this percentage is 20 and 40, respectively. Each relation has one million tuples. For each of the three candidate representations of "now," i.e., `NULL`, Min value, and Max), we have a variant of each table. In the experiments, we have used a composite B-tree index on `V-Begin` and `V-End`, and a B-tree index on `T-Stop` for all tables.

The result is three different tables and three different queries. And each of these exists in three variation corresponding to the three choices for *NOW*. The CPU-times in seconds to answer the queries are shown in Table 1.

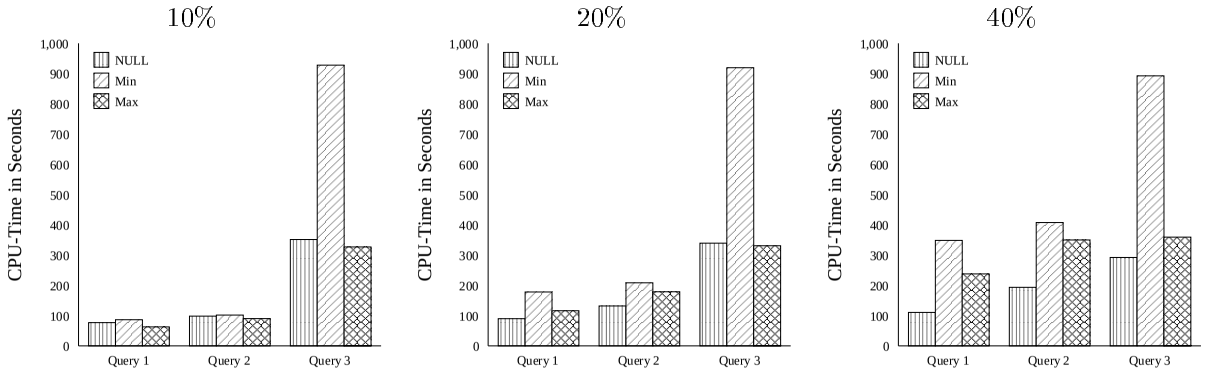

Table 1: CPU-time in Seconds for the Three Queries

It follows that representing "now" by the minimum value is always slowest. When 10% of the tuples are in the current state, it is approximately 5% slower to use `NULL` than the maximum value for the three queries. However, when 20% and 40% of the tuples are current, it is fastest to use `NULL`. It should also be noted that CPU time and elapsed time were almost identical in our experiments (with typical deviations well below 1%).

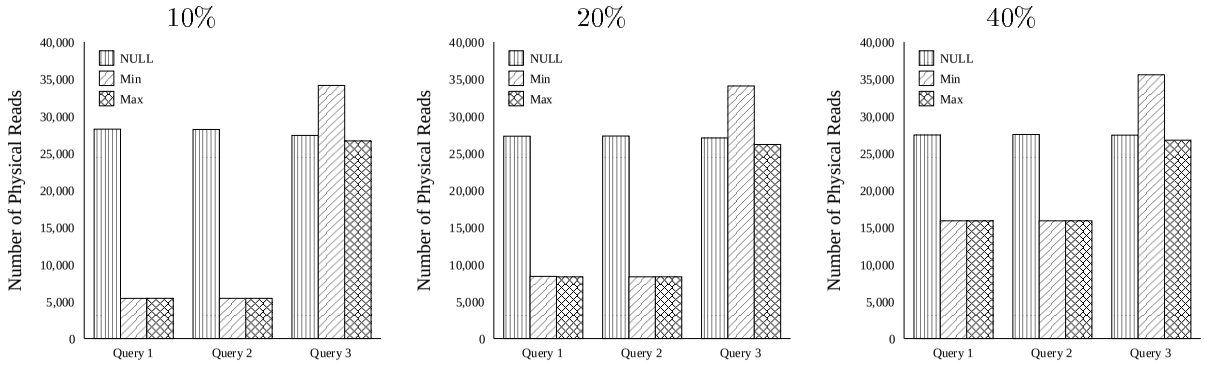

Table 2: Number of Disk Accesses for the Three Queries

Table 2 shows the number of physical disk accesses for the same experiments. Using `NULL` for "now" always results in a full table scan and therefore always results in the same number of pages being read. Using the minimum or maximum value for "now" results in the same number of physical disk accesses for Query 1 and Query 2. The number of disk pages read for Query 3 is independent of the number of

tuples in the current relation (the query does not access the current state), and it is always better to use the maximum value compared to using the minimum value for this query.

Comparing Table 1 to Table 2, there seems to be little correlation between the CPU-time and disk accesses. Query 1 uses little CPU-time and fetches many disk pages using `NULL` for "now"; but it uses more CPU-time for fetching fewer few disk pages when using the maximum value. This phenomenon occurs because it is approximately 3 times slower to compare dates than to compare any other data type in Oracle.

From the experiments, it is clear that the minimum value should not be used to represent "now." We cannot state whether it is best to use `NULL` or the maximum value. If we, e.g., have 10% of the tuples in the current relation it is faster to use the maximum value, but with 40% of the tuples current, it is faster to use `NULL`. We choose to use the maximum value to represent now in the following.

4 Query Processing

This section describes different strategies for processing queries in a layered architecture. The main idea is to reduce product development time. For this purpose, several variants of partial parsers are investigated.

Partial parser approaches are useful in two situations. First, they can significantly reduce the time it takes to release the first version of a new product. Today, this factor often decides whether a product is successful or not. Second, a partial parser approach is useful if many statements of a language are not affected by the (temporal) extension. The parsing of such statement does not have to be implemented, as we will see below.

4.1 A Full Parser

We start with the layered architecture shown in Figure 2. The user enters a query, Q, that is parsed in the layer. Any errors found during parsing are reported. If no errors are found, an equivalent SQL–92 query, called Q', is generated and sent to the DBMS. Query Q can be either an SQL–92 query or a temporal query. During the conversion, the layer uses and possibly updates the metadata maintained by the layer. Finally, it is necessary to do some processing of the output from query Q', e.g., substitute the value representing "now" with the text string "NOW". We call this layered temporal query processing architecture a *full parser architecture*.

With this architecture, it is possible to obtain UC and TUC, and it is possible to process all SQL–92 and temporal queries. Further, all desirable properties of the DBMS can be retained because it is totally encapsulated from the users. Finally, by generating SQL–92 code, the layer can be made platform independent.

As disadvantages, we do not obtain UC with a minimal effort. The SQL–92 parser in the DBMS is not reused; rather, we have to implement it in the layer. This means that before we can start to implement the temporal extensions to SQL–92, we first have to implement SQL–92. Further, SQL–92 queries are unnecessarily parsed twice, once in the layer and once in the DBMS. This performance overhead, we would like to avoid if possible. We thus explore alternative layered architectures next.

4.2 A Partial Parser Architecture

SQL–92 is a large language, making an upward compatible temporal extension even bigger. Because the DBMS has a full SQL–92 parser, it is attractive to only have to implement a parser for the temporal extension in the layer, and to rely on the DBMS's parser for the SQL–92 queries. This idea is illustrated in Figure 7. The parser in the layer is now a *partial parser*—it only *must* know the temporal extensions to SQL–92.

A query Q is entered. If the parser cannot parse Q, it is assumed to be an SQL–92 query and is sent unconverted to the DBMS. If the parsing does not generate an error, Q is a temporal query and is converted to the equivalent SQL–92 query, Q', that is then sent to the DBMS.

This architecture makes it possible to achieve UC with a minimal effort by maximally reusing the underlying DBMS for the processing of SQL–92 queries: All SQL–92 queries will run immediately, and

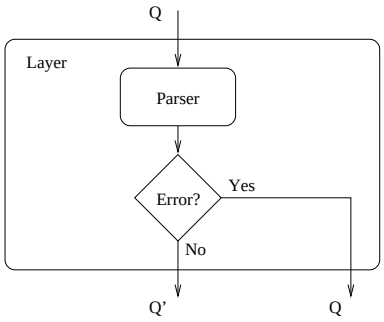

Figure 7: Converting Temporal to SQL–92 Queries with a Partial Parser

error messages to incorrect SQL–92 queries are generated by the DBMS. It is also possible to achieve TUC: If an existing relation is altered to support valid or transaction time, legacy queries using the relation may be detected and modified in the layer.

However, there is still a performance overhead. The layer must start parsing all queries, including SQL–92 queries, and stops only if and when an error is encountered. Further, there is a problem with error handling. The result of an error is that the query is sent to the DBMS, which cannot parse an incorrect temporal query, either. This results in SQL–92 error messages to temporal queries, because the encapsulation of the underlying DBMS is violated.

The source of the disadvantages seems to be that the layer cannot easily and correctly determine whether a query is a temporal or an SQL–92 query. The next architecture attempts to solve this.

4.3 Partial Parser with Optional Hints

With a partial-parser approach with optional hints, the user can indicate whether a query Q is temporal or non-temporal by writing `TEMPORAL` or `PLAIN`, respectively, in a comment before the query. The approach is illustrated in Figure 8.

A query Q is entered. If the scanner finds `PLAIN` in front of the query, it is sent directly to the DBMS. If the scanner finds `TEMPORAL` or no hint, Q is parsed in the layer. If Q is a temporal query, it is converted to Q' which is then sent to the DBMS. If the parser finds an error, the user receives an error message. The presence of `TEMPORAL` indicates that the error is a temporal-query error. Otherwise, the query is assumed to be a SQL–92 query, and it is sent unconverted to the DBMS.

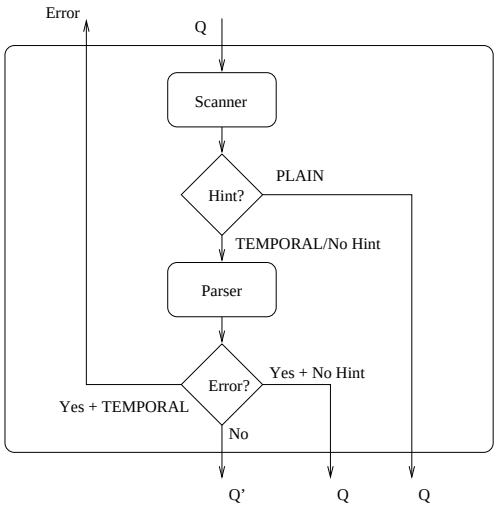

Figure 8: Partial Parser with Optional "Hints"

With this approach, it is possible to achieve UC with a minimal effort, and SQL–92 queries with a hint are parsed only once, leading to faster processing. The architecture also permits for obtaining TUC; and there is good error handling for temporal queries when the `TEMPORAL` hint is used.

However, there are also some problems. Legacy SQL–92 queries are parsed twice if the `PLAIN` hint is not present. Without this hint, we have the same disadvantages as before: a performance overhead for SQL–92 queries and problems with the error handling for temporal queries. We try to eliminate these problems next.

4.4 Partial Parser with Forced "Hints"

With a partial parser approach with forced "hints" temporal queries *must* be tagged with a `TEMPORAL` hint. Thus queries with no hint are assumed to be SQL–92 queries. This way, we are able to distinguish SQL–92 queries from temporal queries without having to revisit legacy code.

The idea is illustrated in Figure 9. When query Q is entered and the scanner does not find `TEMPORAL` in front of the query, it is sent directly to the DBMS. If the scanner finds a `TEMPORAL`, Q is converted to Q', which is then send to the DBMS. If the parser finds an error, this must be a temporal-query error, and an appropriate error message may be generated.

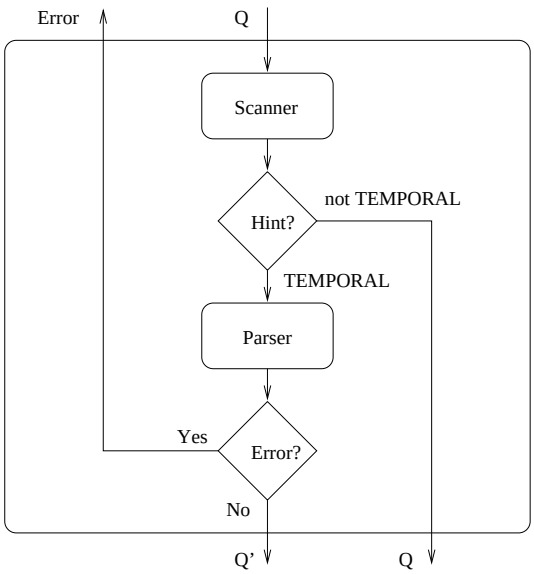

Figure 9: Partial Parser with Forced "Hints"

The advantages of this architecture are the same as for a partial parser with optional hints. We get UC with a minimal effort and fast handling of SQL–92 queries. The disadvantage is that we cannot get TUC. If a table is altered to add temporal support, all legacy queries using the table must be altered by inserting the temporal hint.

4.5 Comparison of Architectures

An overview of the advantages and disadvantages of the different architectures compared to our design goals from Section 2.3 appears in Table 3.

All four architectures are compatible with a platform-independent layer, and they may reuse the components in the DBMS. However, there is less reuse with the full parser. Here we cannot achieve UC with a minimal effort. It is interesting to observe that we cannot obtain both TUC and no performance overhead for SQL–92 queries without revisiting legacy code. For the partial parser with optional hints, we can either achieve TUC or no performance overhead, but not both at the same time. We can retain the desired properties, e.g., error handling, of the DBMS, except for with the partial parser.

	Full parser	Partial parser	Partial parser, optional hints	Partial parser, forced hints
Platform independence	√	√	√	√
Maximum reuse	√	√	√	√
Gradual availability		√	√	√
Upward compatibility		√	√	√
Temporal upward compatibility	√	√	(√)	
Retention of desired properties	√		√	√
Adequate performance			(√)	√

Table 3: Advantages of Different Parser Approaches

The partial parser approaches are consistent with the desire for gradual availability of increasingly more temporal support. The outset is that we want a temporal DBMS that is upward compatible with SQL–92. Then we want to, e.g., have temporal upward compatibility for all non-nested SQL–92 queries, then for all SQL–92 queries, and finally advanced temporal support via new temporal sequenced and non-sequenced queries.

5 Update Processing

Relations supporting transaction time are append-only. This means that, e.g., the logical deletion of tuples from such relations is converted into updates and insertions on the underlying snapshot tables that represent them in the layered temporal DBMS. This section describes how temporal insert, delete, and update statements on bitemporal relations are converted into conventional insert and update statements on the underlying SQL–92 relations. In doing so, we distinguish between TUC, sequenced, and non-sequenced modifications, as defined in Section 2.1.

Below, we illustrate the concepts by means of examples.

5.1 Insertion

The temporal query language is assumed to retain the valid-time periods of tuples as supplied by the user when a tuple is inserted. This means that tuples are inserted "as is"; merging of value-equivalent tuples may be achieved separately, using coalescing.

We illustrate insertion using the TUC statement in Figure 10A, with `Employee` being the relation in Figure 1. Recall that TUC statements are syntactically correct SQL–92 statements, but are applied to temporal relations. TUC insertions must timestamp inserted tuples so that they remain in the current state (in both valid time and transaction time) starting at the time of the insertion. That way, other TUC statements such as queries and constraints may simply be implemented by timeslicing all argument temporal tables as of the current time (again, in both valid time and transaction time) and then evaluate the (SQL–92) query on the snapshots.

```
A     INSERT INTO Employee          B     INSERT INTO Employee VALUES
      VALUES ('Kim', 'Shoe')              ('Kim', 'Shoe', CURRENT_TIME, 31-12-9999,
                                                  CURRENT_TIME, 31-12-9999)
```

Figure 10: A TUC Insertion

The sample modification inserts an employee Kim who is in the Shoe department. The temporal insert statement is converted into the SQL–92 insert statement in Figure 10B. It assigns the current time to transaction-time start and valid-time begin, and `31-12-9999` (the maximum date, denoting an unbound current time variable) to transaction-time stop and valid-time end. This way, the tuple inserted will always qualify for queries on the current state (and not earlier) and is thus a correct TUC insertion of a tuple.

Sequenced and non-sequenced insertions are handled similarly (they are exemplified in the next sections, covering deletion and update). However, these insertions include a specification of the valid-time period to be associated with the inserted tuples. The transaction-time timestamps are handled by the layer, as shown above.

5.2 Deletion

A temporal deletion statement includes a predicate that specifies which tuples to delete. TUC deletions delete tuples so that they no longer appear in the current transaction-time and valid-time states.

Sequenced and non-sequenced deletions include a period specifying for which valid-time period to delete the qualifying tuples (from the current transaction-time state). Such a deletion is converted into an SQL–92 update of the transaction-time stop attribute values to the current time. This is followed by from zero to two insertions, to reflect the changes to the time attributes. The different numbers of insertions are due to three different cases which may occur.

- The valid-time period of a tuple to be deleted is totally contained in the period specified in the temporal delete statement. In this case, no insertions are needed.
- The valid-time period a tuple to be deleted overlaps with the period specified in the delete statement, and the part not overlapped by the specified period is a single period. In this case, one insertion is needed.
- The valid-time period a tuple to be deleted overlaps with the period specified in the delete statement, and the part not overlapped by the specified period consists of two periods. In this case, two insertions are needed.

As an example that illustrates the latter case, consider the sequenced deletion in Figure 11A which deletes Alex from the Sports department in the period [25-8-1996, 30-8-1996). (Figure 1 shows that Alex is currently registered as being in Sports from 23-8-1996 to 31-8-1996.)

```
A   SEQUENCED VALID PERIOD '25-8-1996 - 30-8-1996'
    DELETE FROM Employee
    WHERE Name = 'Alex' AND Dept = 'Sports'

B   UPDATE Employee SET T-Stop = CURRENT_TIME
    WHERE Name = 'Alex' AND   Dept = 'Sports' AND T-Stop = 31-12-9999
    INSERT INTO Employee VALUES ('Alex', 'Sports', CURRENT_TIME, 31-12-9999,
                                                   23-8-1996, 25-8-1996)
    INSERT INTO Employee VALUES ('Alex', 'Sports', CURRENT_TIME, 31-12-9999,
                                                   30-8-1996, 31-8-1996)
```

Figure 11: A Sequenced Deletion

The statement is converted to the SQL–92 update and inserts in Figure 11B. First, the tuple to be deleted from is logically deleted by setting the transaction-time stop to the current time. Because the period to delete, [25-8-1996, 30-8-1996), splits the old valid-time period, [23-8-1996, 31-8-1996), into two periods, two tuples must be inserted to record this information. The first insertion covers the valid-time period [23-8-1996, 25-8-1996), and the second insertion covers the period [30-8-1996, 31-8-1996). The transaction-time start of both insertions are the current time, and their transaction-time stop is 31-12-9999.

Non-sequenced deletions are handled similarly to sequenced deletions, and TUC deletions are handled by setting transaction-time stop values to the current time.

5.3 Update

In a snapshot relation, an update can be modeled as a deletion followed by an insertion. The same idea applies to a bitemporal relation. This means that the tuples to be updated are first logically deleted,

by setting their transaction-time stop to the current time. Then from one to three insertions follow, to reflect the new values and changes to the time attributes (the same three cases as for deletion must be considered).

As an example, consider the non-sequenced update in Figure 12A which updates the `Employee` relation (in Figure 1) to record that Alex is in the Shoe department between August 30 and September 10, 1996.

```
A   NONSEQUENCED VALID PERIOD '30-8-1996 - 10-9-1996'
    UPDATE Employee SET Department = 'Shoe'
    WHERE Name = 'Alex'

B   UPDATE Employee SET T-Stop = CURRENT_TIME
    WHERE Name = 'Alex' AND T-Stop = 31-12-9999
    INSERT INTO Employee VALUES ('Alex', 'Sports', CURRENT_TIME, 31-12-9999,
                                                    23-8-1996, 30-8-1996)
    INSERT INTO Employee VALUES ('Alex', 'Shoe', CURRENT_TIME, 31-12-9999,
                                                  30-8-1996, 10-9-1996)
```

Figure 12: A Non-sequenced Update

The update is converted to the SQL–92 update and insertions in Figure 12B. First, we logically delete the tuple to update by setting the transaction-time stop attribute to `current_time`. Because the update specifies a valid-time period that overlaps the valid-time period of the tuple in the database, we first insert a tuple to reflect that Alex is not in the Sports department as long as first recorded, i.e., only from August 23 to August 30. Then we insert a tuple with the new department and valid-time period.

Sequenced updates are handled similarly to non-sequenced updates, with the exception that the valid-time periods attached to the tuples resulting from the update are derived from the information in the database, not just the valid-time period specified by the user. Specifically, the resulting times of tuples are found as the intersections between the existing tuples and the time period specified in the query.

A TUC update is modeled as a TUC deletion followed by a TUC insertion, and it results in one tuple being inserted.

6 Transaction Processing

In this section, we discuss how to implement ACID properties [12] of transactions in the layer by exploiting the ACID properties of the DBMS. Specifically, we show how concurrency control and recovery mechanisms can be implemented using the services of the DBMS. Next, the correct timestamping of database modifications is explored. Finally, we study how the granularity of timestamps affects the correctness of transaction processing.

6.1 ACID Properties of Transactions

One of our design goals is to retain the desirable properties of the underlying DBMS. The ACID properties of transactions are examples of such desirable properties.

The ACID properties of temporal SQL transactions are retained by mapping each temporal transaction to a single SQL–92 transaction. The alternative of allowing the layer to map a temporal SQL transaction to several SQL–92 transactions, while easing the implementation of temporal SQL transactions, leads to hard-to-solve problems.

To illustrate, assume that a temporal SQL transaction is mapped to two SQL–92 transactions. During execution it may then happen that one SQL–92 transaction commits but the other fails, meaning that the temporal SQL transaction fails and should be rolled back. This, however, is not easily possible—other (e.g., committed) transactions may already have seen the effects of the committed SQL–92 transaction.

Next, it is generally not sufficient to simply require that each temporal SQL transaction is mapped to a single SQL–92 transaction. It must also be guaranteed that the SQL–92 transaction does not contain DDL statements. This is so because the SQL–92 standard permits DDL statements to issue implicit

commits [17, p. 76]. Thus the SQL–92 transaction becomes several SQL–92 transactions, yielding the same problem as before.

The conclusion is that the ACID properties of temporal SQL transactions are guaranteed if the SQL–92 transactions satisfy the ACID properties and if we map each temporal SQL transaction to exactly one SQL–92 transaction that does not contain DDL statements.

6.2 Timestamping Database Updates—Correctness and Efficiency

When supporting transaction time, all previously current database states are retained. Each update transaction transforms the current database state to a new current state. In practice, this is achieved by associating a pair of an insertion and a deletion time with each tuple. These times are managed by the DBMS, transparently to the user. The insertion time of a tuple indicates when the tuple became part of the current state of the database, and the deletion time indicates that the tuple is still current or when it ceased to be current.

To ensure that the database correctly records all previously current states, the timestamps given to tuples by the transactions must satisfy four requirements. First, all insertions into and deletions from the current state by a transaction must occur simultaneously, meaning that the insertion times of insertions and the deletion times of deletions must all be the same time. If not, we may observe inconsistent database states. For example, if the two updates in a debit-credit transaction are given different timestamps and we inspect the database state current between the two timestamps, we see an inconsistent state. Second, the transactions cannot choose their timestamp times arbitrarily. Rather, the times given to updates by the transactions must be consistent with a serialization order of the transactions. Thus, if transaction T_1 uses timestamp t_{T_1} and transaction T_2 uses timestamp t_{T_2}, with $t_{T_1} < t_{T_2}$, then there must exist a serialization order in which T_1 is before T_2. Third, a transaction cannot choose as its timestamp value a time that is before it has taken its last lock. If this restriction is not met, queries may observe inconsistent database states. Fourth, it is our contention that it is undesirable that a transaction uses a timestamp value that is after its commit time. This would result in "phantom changes" to the database, i.e., "changes" that occur when no transactions are executing.

Using the (ready-to) commit time of each transactions for its timestamps is a simple and obvious choice that satisfies all the requirements. Salzberg [19] has previously studied two approaches to timestamping with this choice.

In the first approach, all updates by a transaction are *deferred* until it has acquired all its locks. It is a serious complication that it may not be possible to determine that a transaction has taken all the locks it needs before the transaction is ready to commit (c.f. practical two-phase locking where locks are not released until the transaction is ready to commit). Next, it is a problem with this approach for a transaction to read its own updates. Thus, this approach is only suitable for short and simple transactions.

The second approach is to *revisit* and timestamp all the tuples after all locks have been acquired, i.e., in practice when the transaction is ready to commit. This approach is recommended because it is general and guarantees correctness. The cost is to have to visit tuples twice: once to write a temporary value for the time attributes, and once to update the temporary value to the commit time. This cost is affected by the hit ratio for the buffer of tuples to revisit. With a high hit ratio, tuples to revisit and give the correct timestamp are often in the buffer; with a low hit ratio, tuples have often to be fetched twice from disk.

In order to avoid some of the overhead of the basic timestamp-after-commit scheme, we propose an improved approach where tuples are timestamped *at first update*. This approach trades correctness for performance: it generally does not satisfy the third requirement from above. This does not render the approach useless, but it may not be applicable for all applications (c.f., SQL–92's Transaction Isolation Levels [17, pp. 293–302]). In the presentation that follows, we disregard the third requirement.

The approach is an optimistic one. We select the time of the first update, t^s_T, of a transaction, T, as the transaction's timestamp time, hoping that we will be able to use this time for timestamping all updates without violating the second requirement from above. If the transaction has only the one update, the chosen timestamp time satisfies correctness. However, each update that the transaction makes may, or may not, invalidate our choice of timestamp time.

Consider a tuple x inserted into the current state of the database by a transaction T' and at time $t^s_{T'}$ and assume that T is to update this tuple. As T sees a result of T', T' must be before T in any serialization order. The second requirement then implies that the timestamp time of T' must be before the timestamp time of T, i.e., it is required that $t^s_{T'} < t^s_T$. When the update is to be carried out, this condition is checked. If it is satisfied, our choice of timestamp time for T does not violate correctness, and the update is carried out using time t^s_T. Subsequent updates are then processed similarly. If the condition is not satisfied, the choice of timestamp time does violate correctness, and we say that the two involved transactions conflict. In this case, timestamp-after-commit is used. If all updates satisfy the requirement, the choice of timestamp satisfies the serializability requirement, and transaction T can simply commit without having to revisit any tuples.

This new scheme has other notable characteristics. The first update will never lead to a conflict. This is so because t^s_T will be larger than the time when we acquire a write lock on the tuple to update. This time, in turn, will be larger than the timestamp of the tuple, $t^s_{T'}$. Thus, transactions with a single update will never experience a conflict.

Next, observe that using the time of the first update for timestamping makes the chance of conflicts between concurrent transactions the smallest possible. It is also not necessary to attempt to determine when in a transaction all locks have been acquired.

In both the timestamp-after-commit and timestamp-at-first-update, it is necessary for a transaction to retain a list of updated tuples until the transaction is ready to commit. With the timestamp-at-first-update there is an overhead of one comparison for each tuple to update. However, the comparisons are on tuples that have already been fetched in order to do the update.

The benefit of using timestamp-at-first-update compared to using timestamping-after-commit thus are that when there are no conflicts, we do not have to revisit updated tuples to update their timestamp when the transaction is ready to commit. When there are conflicts the two timestamp algorithms are virtually identical.

To summarize, the general approach we propose for timestamping is as follows. A temporal SQL transaction is mapped to a single SQL–92 transaction without DDL statements. The serialization level for the SQL–92 transaction is set to "serializable." All timestamp of tuples written by the SQL–92 transaction are given the time of the first update as their value, and the identity of each updated tuple is recorded. More precisely, the time of the first update for a transaction is the time of the system clock when the first tuple to be updated has been locked. When the transaction is ready to commit and if there were any conflicts, the update-after-commit procedure is evoked; otherwise, the SQL–92 transaction commits.

We have chosen to eagerly update timestamps with (possibly) temporary values. When a transaction is ready to commit, and if there were any conflicts, we update the temporary value to the (ready to) commit time of the transaction. An alternative is to update timestamps to reflect their correct values lazily, i.e., as needed. However, this requires that information be retained for each tuple telling which transaction updated it and that the commit time of all transactions also be retained. The lazy approach thus adds complexity over the eager approach. To make the layer as thin as possible we choose the eager approach.

Recovery is an important part of a DBMS that normally is transparent to end users. When constructing the layered approach, we are not different from end users and can rely on the recovery mechanisms implemented in the DBMS. We see no reason why recovery should be faster or slower using a layered approach.

6.3 Granularity of Timestamps

Through out this paper we have used time stamps with a granularity of days. It is chosen for illustrative purposes, if actually used in a temporal database it may result in incorrect results being returned to the users.

To exemplify this, we consider an example. We have a transaction-time relation `Employee` where we record the names and departments of employees. The data is stored at a granularity of a *day*! Consider the following scenario.

```
Transaction 1              Queries                          Transaction 3
(at 9.00 a.m.)
INSERT INTO Employee
VALUES ('Tom', 'Shoe')
COMMIT
                           (at 11.00 a.m.)
                           SELECT Name
                           FROM Employee
                           WHERE Department = 'Shoe'
                                                            (at 1.00 p.m.)
                                                            DELETE FROM Employee
                                                            WHERE Name = 'Tom'
                                                            COMMIT
                           (at 3.00 p.m.)
                           SELECT Name
                           FROM Employee
                           WHERE Department = 'Shoe'
```

The query at 11.00 a.m. returns that Tom is in the Shoe department; then at 3.00 p.m. the same query returns that Tom is not in the Shoe department. The two queries are identical and access the same state of the transaction-time relation. Therefore, they should return the same result.

The example illustrates a problem that may occur when the granularity of transaction time is too coarse. To ensure correct queries, the granularity of transaction time must be chosen so fine that the following cannot happen within a single granule.

1. One transaction inserts a tuple and commits,
2. a query retrieves the tuple, and
3. another transaction delete the same tuple and commits.

This problem can be avoided by not committing a transaction until the start of the next granule. In the example, Transactions 1 and 2 should thus not be permitted to commit until the next day. This is the technique we will apply in the layer. The default granularity of the data type `TIMESTAMP` in SQL–92 is adequately fine—with seconds being the unit, there are by default six digits after the decimal period.

A special case of the scenario just described occurs if tuples are TUC inserted and TUC deleted within the same transaction. They will then be timestamped with the same valid-time begin, valid-time end, transaction-time start and transaction-time stop values. Such tuples cannot "harm" other transactions as above, and the tuples will never affect the result of any TUC or sequenced statements.

Therefore, one could argue that such tuples should be physically deleted from a bitemporal database. However, we choose not to do this because it violates the append-only nature of a bitemporal database. Further, it is a faithful recording of what actually happens.

7 Conclusion and Future Research

We have investigated concepts and techniques for implementing a temporal SQL using a layered approach where the temporal SQL is implemented via a software layer on top of an existing DBMS. The layer reuses the functionality of the DBMS in order to support aspects such as access control, query optimization, concurrency control, indexing, storages, etc.

While developing a full-fledged DBMS that supports a superset of SQL is a daunting task that only the major vendors can expect to accomplish, this layered technology promises much faster development. Assuming that the underlying DBMS is an SQL–92 compliant black box makes this technology inherently open and technology transferable. It may be adopted by a much wider range of smaller software vendors that would like to provide more advanced database functionality than offered by current products.

We stressed seven design goals, namely upward compatibility with a minimal coding effort, gradual availability of temporal functionality, temporal upward compatibility, maximum reuse, retention of

desired properties of the DBMS, platform independence, and adequate performance; and we favored achieving a thin layer over high performance.

With these goals in mind, we explored what we believe to be the central issues in the layered implementation of temporal functionality on a relational SQL–92 platform. We first considered the options for the domain of timestamps, and for representing the temporal database variable "now" within the chosen domain. Both the storage of "now" in databases and the implications for queries of its use were studied. To set apart the possibilities for "now," their performance characteristics were compared. Then followed an exploration of different query processing architectures. We showed how the partial-parser architecture may be used for achieving upward compatibility with a minimal effort and for satisfying additional goals. We then described how update processing is handled in the layer. Finally, we considered the processing of temporal transactions. We sketched how ACID-properties of temporal transactions are retained using the non-temporal ACID transactions of the underlying DBMS, and we also covered how to correctly timestamp the results of update transactions.

This work points to several directions for future research. First, a more comprehensive study of the performance characteristics of layered implementation of temporal functionality is warranted. Second, we feel that the relative merits of the strict preprocessor approach adopted here should be compared with those of an architecture where the layer is extended with a control component that controls, e.g., the execution order of statement sequences and the checking of database consistency. Third, we believe that it would be interesting to study hybrid architectures, in-between the conventional integrated architecture of current DBMS produces and the preprocessor approach studied here. A hybrid architecture should be able to exploit temporal implementation techniques, e.g., indices and join algorithms, while also reusing the services of an SQL–92 DBMS. Finally, it may be of interest to try to apply layered techniques to other types of relational extensions.

References

[1] I. Ahn and R. Snodgrass. Partitioned Storage for Temporal Databases. *Information Systems*, 13(4):369–391, 1988.

[2] I. Ahn and R. Snodgrass. Performance Analysis of Temporal Queries. *Information Systems*, 49:103–146, 1989.

[3] D. S. Batory, J. R. Barnett, J. F. Garza, K. P. Smith, K. Tsukuda, B. C. Twichell and T. E. Wise. Genesis: An Extensible Database Management System. In S. B. Zdonik and D. Maier, editors, *Readings in Object-Oriented Database Systems*, Chapter 7.4, pages 500–518. Morgan Kaufmann Publishers, 1990.

[4] M. H. Böhlen. *The Temporal Deductive Database System Chronolog.* Ph.D. thesis, Departement Informatik, ETH Zurich, 1994.

[5] M. Böhlen, C. S. Jensen, and R. T. Snodgrass. Evaluating and Enhancing the Completeness of TSQL2. Technical Report TR 95-05, Department of Computer Science University of Arizona, Tucson, AZ 85721, June 1995.

[6] Michael Böhlen. Temporal Database System Implementations. *ACM SIGMOD Record*, 24(4), December 1995.

[7] M. Carey, D. J. DeWitt, G. Graefe, D. M. Haight, J. E. Richardson, D. T. Schuh, E. J. Shekita, and S. L. Vandenberg. The EXODUS Extensible DBMS Project: An Overview. In S. B. Zdonik and D. Maier, editors, *Readings in Object-Oriented Database Systems*, Chapter 7.3, pages 474–499. Morgan Kaufmann Publishers, 1990.

[8] J. Clifford and A. Tuzhilin, editors. *Recent Advances in Temporal Databases.* Workshops in Computing Series, Springer-Verlag, November 1995, ISBN 3-540-19945-4.

[9] C. J. Date. A Proposal for Adding Date and Time Support to SQL. *ACM SIGMOD Record*, 17(2):53–76, November 1998.

[10] C. Davies, B. Lazell, M. Hughes, and L. Cooper. Time is Just Another Attribute—or at Least, Just Another Dimension, pages 175–193. In [8]

[11] C. E. Dyreson and R. T. Snodgrass. *A Timestamp Representation*, Chapter 25, pages 475–499. In [27].

[12] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann Publishers, 1993.

[13] G. M. Lohman, B. Lindsay, H. Pirahesh, and K. B. Schiefer. Extensions to Starburst: Objects, Types, Functions, and Rules. *Communication of the ACM*, 34(10):94–109, October 1991.

[14] C. S. Jensen, J. Clifford, R. Elmasri, S. K. Gadia, P. Hayes, and S. Jajodia, editors. A Glossary of Temporal Database Concepts. *ACM SIGMOD Record*, 23(1):52–64, March 1994.

[15] A. M. Keller. Penguin: Objects for Programs, Relations for Persistence. URL: http://www-db.stanford.edu/pub/keller/1994/Penguin-overview-paper.ps, April 1994.

[16] T. Y. C. Leung and H. Pirahesh. Querying Historical Data in IBM DB2 C/S DBMS Using Recursive SQL, pages 315–331. In [8].

[17] J. Melton and A. R. Simon. *Understanding the New SQL: A Complete Guide.* ISBN 1-55860-245-3. Morgan Kaufmann Publishers, 1993.

[18] Oracle Corp. *Oracle7 Server Concepts Release 7.2*, March 1995.

[19] B. Salzberg. Timestamping After Commit. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, pages 160–167, Austin, TX, September 1994.

[20] U. Schreier, H. Pirahesh, R. Agrawal, and C. Mohan. Alert: An Architecture for Transforming a Passive DBMS into an Active DBMS. *Proceedings of the VLDB Conference*, pages 469–478, 1991.

[21] A. R. Simon. *Strategic Database Technology: Management for the Year 2000.* ISBN 1-55860-264-X. Morgan Kaufmann Publishers, 1995.

[22] R. T. Snodgrass. The Temporal Query Language TQuel. *Transaction on Database Systems*, 12(2):247–298, June 1987.

[23] R. T. Snodgrass. *An Overview of TQuel*, Chapter 6, pages 141–182. In [33].

[24] R. T. Snodgrass and I. Ahn. Temporal Databases. *IEEE Computer*, 19(9):35–42, September 1986.

[25] R. T. Snodgrass. *Event Tables*, Chapter 16, pages 311–318. In [27].

[26] R. T. Snodgrass. Change Proposal to SQL/Temporal: A Road Map of Additions to SQL/Temporal. ANSI Expert's Contribution ANSI X3h2-96-013 ISO/IEC JTC1/SC21/WG3 DBL ??, February 1996.

[27] R. T. Snodgrass, editor. *The TSQL2 Temporal Query Language.* ISBN 0-7923-9614-6. Kluwer Academic Publishers, 1995.

[28] A. Steiner, M. Böhlen, C. S. Jensen, and R. Snodgrass. Implementation of TimeDB. URL: http://www.iesd.auc.dk/general/dbs/tdb/timecenter/software/timedb.tar.gz, 1995.

[29] M. Stonebraker, L. A. Rowe, and M. Hirohama. The Implementation of Postgres. *IEEE Transaction on Knowledge and Data Engineering*, 2(1):125–142, March 1990.

[30] M. Stonebraker and L. A. Rowe. The Design of Postgres. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 340–355, May 1986.

[31] M. Stonebraker and G. Kemnitz. The Postgres Next-generation Database Management System. *Communication of the ACM*, 34(10):78–92, October 1991.

[32] T. Takahashi and A. M. Keller. Implementation of Object View Query on a Relational Database. In *Data and Knowledge Systems for Manufacturing and Engineering*, May 1994.

[33] A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, editors. *Temporal Databases: Theory, Design, and Implementation.* Database Systems and Applications Series. Benjamin/Cummings, Redwood City, CA, 1993.

[34] C. Vassilakis, N. Lorentzos, and P. Georgiadis. Transaction Support in a Temporal DBMS, pages 255–271. In [8].

A Temporal Queries

The appendix lists the actual queries used to determine which value to use for *NOW* in Section 3.4. There, we wanted to use a SQL–92 `CASE` statement in the queries, but this construct is not available in the DBMS used, Oracle 7.2.2.4. Instead, have we used `NVL` when using `NULL` for *NOW*; this operator is similar to the SQL–92 `COALESCE` operator. When we studied the performance of using the minimum and the maximum values for *NOW*, we used `OR` in place of the `CASE` statement.

In the test, we have used the Oracle data type `DATE` for time attributes using the finest granularity of seconds. The minimum and maximum values for this data type are `'01-01--4712 00:00:00'` and `'31-12-4712 23:59:59'`, respectively. Finally, because Oracle does not allow the use of hyphens in identifiers, we use in the queries `VTS` for `V-Begin`, `VTE` for `V-End`, `TTS` for `T-Start`, and `TTE` for `T-Stop`.

All queries have the same `SELECT` and `FROM` clauses, but have different predicates in their `WHERE` clauses. This general format and the varying predicates follow.

```
SELECT Unique1, two
FROM   TableOne
WHERE  <predicate>;
```

Query 1 predicates using `NULL`, the minimum value, and the maximum value for *NOW*:

```
TS <= SYSDATE AND NVL(VTE, SYSDATE) >= SYSDATE AND TTE IS NULL

VTS <= SYSDATE AND (VTE > SYSDATE OR VTE = '01-01--4712 00:00:00') AND
TTE  = '01-01--4712 00:00:00'

VTS <= SYSDATE AND VTE >= SYSDATE AND TTE  = '31-12-4712 23:59:59'
```

Query 2 predicates using `NULL`, the minimum value, and the maximum value for *NOW*:

```
VTS <= '01-01-1994 00:00:00' AND NVL(VTE, SYSDATE) > '01-01-1994 00:00:00' AND
TTE IS NULL

VTS <= '01-01-1994 00:00:00' AND
(VTE > '01-01-1994 00:00:00' OR VTE = '01-01--4712 00:00:00') AND
TTE = '01-01--4712 00:00:00'

VTS <= '01-01-1994 00:00:00' AND VTE > '01-01-1994 00:00:00' AND
TTE =  '31-12-4712 23:59:59'
```

Query 3 predicates using `NULL`, the minimum value, and the maximum value for *NOW*:

```
VTS <= '01-01-1993 00:00:00' AND NVL(VTE, SYSDATE) > '01-01-1993 00:00:00' AND
TTS <= '01-01-1993 00:00:00' AND NVL(TTE, SYSDATE) > '01-01-1993 00:00:00'

VTS <= '01-01-1993 00:00:00' AND
(VTE > '01-01-1993 00:00:00' OR VTE = '01-01--4712 00:00:00') AND
TTS <= '01-01-1993 00:00:00' AND
(TTE  > '01-01-1993 00:00:00' OR TTE = '01-01--4712 00:00:00')

VTS <= '01-01-1993 00:00:00' AND VTE > '01-01-1993 00:00:00' AND
TTS <= '01-01-1993 00:00:00' AND TTE  > '01-01-1993 00:00:00'
```