Proceedings of the Fifth International Conference on Database Systems for Advanced Applications
Melbourne, Australia, pp. 371-380, 1-4 April, 1997

Layered Temporal DBMS's—Concepts and Techniques

Kristian Torp *Christian S. Jensen* *Michael Böhlen*
Department of Computer Science, Aalborg University
Fredrik Bajers Vej 7E, DK-9220 Aalborg Ø, DENMARK
{torp,csj,boehlen}@cs.auc.dk

Abstract

A wide range of database applications manage time-varying data, and it is well-known that querying and correctly updating time-varying data is difficult and error-prone when using standard SQL. Temporal extensions of SQL 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. In contrast, this paper explores the concepts and techniques involved in implementing a temporally enhanced SQL while maximally reusing the facilities of an existing SQL implementation. The topics covered span the choice of an adequate timestamp domain that includes the time variable "NOW," a comparison of query processing architectures, and transaction processing, the latter including how to ensure ACID properties and assign timestamps to updates.

Keywords Temporal databases, SQL, layered architecture, legacy issues

1 Introduction

A wide variety of existing database applications manage time-varying data (e.g., see [8, p. 670] [13]). 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 [6, 9, 16]). 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., [17, 19]) 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 a language permits 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* also used for implementing commercial relational DBMS's (see, e.g., [1, 18, 19]). 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. 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).

This paper explores the implementation of temporal query languages using a *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.

With this architecture there is a potential for reusing the services of the underlying DBMS, e.g., the concurrency control and recovery mechanisms, for implementing the extended functionality, and upward compatibility may potentially be achieved with a minimal coding effort. The major disadvantages are the entry costs that a DBMS imposes on its clients, as well as the impossibility of directly manipulating DBMS-internal data structures.

Proceedings of the Fifth International Conference on Database Systems for Advanced Applications, Melbourne, Australia, April 1–4, 1997.

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 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, and we discuss issues involved in implementing temporal-transaction processing.

A partial parser has been implemented that provides a minimum of temporal support. We have chosen a commercial DBMS as the underlying DBMS, and not an extensible system (Chapter 7 of reference [21] covers several such systems) because we want to investigate the seamless migration of legacy systems.

Little related work has appeared that considers a layered implementation of temporal query languages. The research reported by Vassilakis et al. [20] 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 [10, 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.

Recently, a layered implementation, based on Oracle, was pursued in the TIMEDB prototype that implements the ATSQL query language [3], which incorporates ideas from TSQL2 [17] and ChronoLog [2]. The topics covered in this paper are partly inspired by and generalize TIMEDB.

Most recently, Finger and McBrien [7] studied which value to use for *NOW* in the valid-time dimension when transactions are taken into consideration. But their work, which focuses on valid time and has a less practical orientantion, covers issues largely orthogonal to those addressed here. Among the most significant differences, they do not consider representational issues and do not use timestamping-after-commit [12], as done here.

The rest of the paper is organized as follows. Section 2 characterizes temporal support, introduces the layered architecture, states design goals, and touches upon the limitations of the general approach. Section 3 is concerned with the domain of timestamps. Different parser architectures for temporal query processing are the topic of Section 4, and Section 5 is devoted to the processing of temporal transactions. Finally, Section 6 concludes and points to research directions.

2 Temporal SQL and Design Goals

Following an introduction to 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 [15]. 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*; and if neither time is supported, it is termed *non-temporal.*

The relation `Employee` in Figure 1 is an example of a bitemporal relation. 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.

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, on 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) [4] and snapshot reducibility (SR) [14]. 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 [4]. 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

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

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). 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 applications 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 of application programmers using SQL–92 should be applicable to the added temporal functionality, making it easier to understand and use the new facilities.

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 satisfy snapshot reducibility with respect 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.

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.

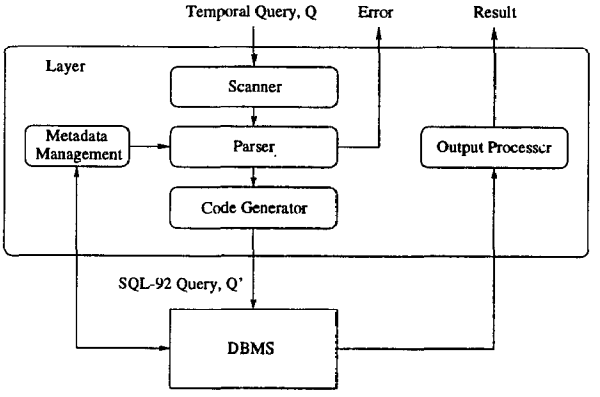

Figure 2: The Layered Temporal Database Architecture

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 following sequenced temporal query that finds the name and department of employees in the sports department and how long they have been there.

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

The new keywords SEQUENCED VALID indicate that the query should be computed over all (valid) times, but just for the current (transaction time) state. To convert this query to SQL–92, the SEQUENCED VALID is omitted, attributes V-Begin and V-End are added to the SELECT clause, and the WHERE clause is extended with conditions on T-Start and T-Stop to ensure that only current tuples are considered. If evaluated on August 25, the following table results.

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

The temporal query is written in the temporal query language ATSQL [3]. It is beyond the scope of this paper to define the syntax and semantics of this language. However, the extensions are consistent with SQL–92 and are easy to understand.

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 desirable 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 any particular 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 temporal databases should be processed as fast as the corresponding SQL–92 queries on the corresponding snapshot database (i.e., with the same information, but using explicit time attributes).

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

An important question when adopting a layered architecture is whether it is practical or even possible to translate all temporal SQL queries to SQL–92 queries. 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 a systematic mapping is not available.

3 Representing the Time Domain

As illustrated in Figure 1, four extra attributes, termed timestamp attributes are used when recording the temporal aspects of a tuple. Next, we will discuss which domain to use for the timestamp attributes and how to represent the special temporal database value "*NOW*."

3.1 Choosing a 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 [10].

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 a much wider range of times with a finer precision. 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, e.g, 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.

As an example of these problems, notice that adding one month to a date depends on which month the date is in. The addition routine must add 31 days to a March date and 30 days to an April date. Thus, addition must be performed in the layer. Indeed, the manipulation of time attributes must to a large extent 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.

We have here reached one of the limitation on building on top of an existing DBMS: Adding a new data type in the layer is a major modification when the underlying DBMS does not support abstract data types. In conclusion, we recommend using the built-in data type `TIMESTAMP` for timestamp attributes.

3.2 Representing *NOW*

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, specifically either the smallest or the largest `TIMESTAMP` value. 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 [11], 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 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 each of `NULL`, the minimum value, and the maximum value as *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 = <now rep.>
  THEN CURRENT_TIMESTAMP
  ELSE relation-name.V-End
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*. Next, we compare their performance. Specifically, for each choice for *NOW*, we perform each of three different representative queries on three different relations. We consider timeslice queries because of their importance in temporal query languages [19]. The queries favor the current state, which is assumed to be accessed much more frequently than old states.

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 that end at *NOW* thus qualify. 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 time and thus retrieves a past belief about a past state of reality.

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

[1] When using `NULL`, the shorter conditional value expression `COALESCE` may also be used.

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. There are thus three different tables and three different queries; and each combination of a table and a query exists in three variations, one for each choice for *NOW*.

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 CPU-times in seconds to answer the queries are shown in Table 1. The tests were performed on a SUN Sparc 10 using the Oracle RDBMS version 7.2.2.4.

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.

The next step is to consider the number of physical disk reads. Using NULL always results in a full table scan. Using the maximum value, the number of physical disk reads increases with the percentage of tuples in the current state. In the case of 40% of the tuples in the current state, the number of physical disk reads is approximately the same when using either of NULL and the maximum value. Using the minimum value performs similar to using the maximum value, except in the case of 40% current tuples where using the minimum value results in 35% more physical disk reads.

Based on the analysis above, we choose to use the maximum value for representing *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 statements does not have to be implemented, as we will see.

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.

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 3. 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.

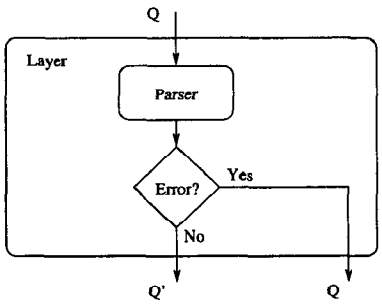

Figure 3: Partial Parser

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 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.

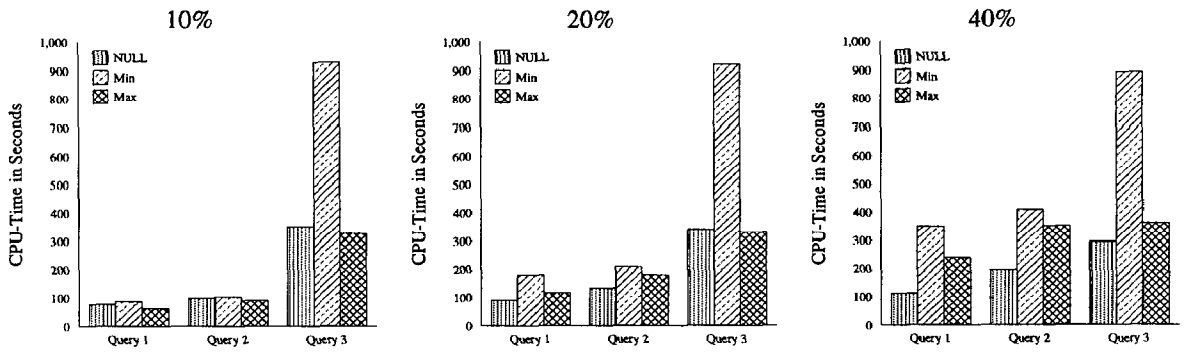

Table 1: CPU-time in Seconds for the Three Queries

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.

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—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 4.

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.

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.

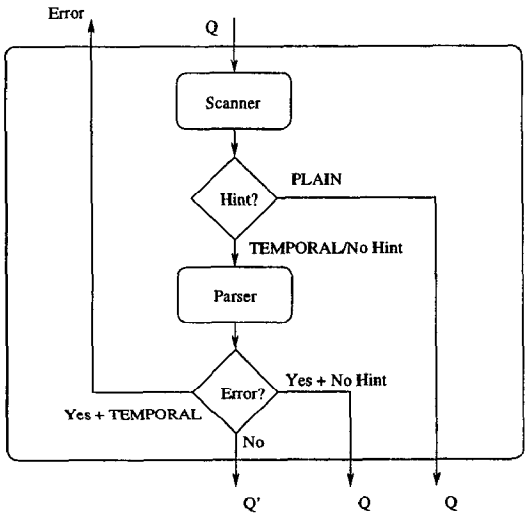

Figure 4: Partial Parser with Optional "Hints"

4.4 Partial Parser—Enforced "Hints"

With a partial parser approach with enforced "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 5. 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 sent to the DBMS. If the parser finds an error, this must be a temporal-query error, and an appropriate error message may be generated.

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.

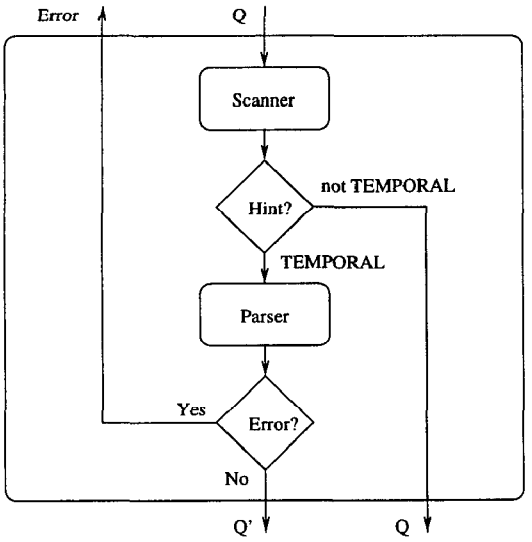

Figure 5: Partial Parser with Enforced "Hints"

4.5 Comparison of Architectures

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 in the case of the partial parser.

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 Transaction Processing

In this section, we discuss how to implement ACID properties [8] 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. Finally, the effective timestamping of database modifications is explored.

5.1 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 [10, p. 76]. Thus the SQL–92 transaction becomes several SQL–92 transactions, yielding the same problem as before.

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.

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.

5.2 Timestamping of Updates

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 system 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 may be 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 the requirements. Salzberg [12] has previously studied two approaches to implementing this choice of timestamping.

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 (cf., practical two-phase locking). 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 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 dependent on the hit ratio for the buffer of tuples to revisit.

In order to avoid some of the overhead of the basic timestamp-after-commit scheme, we propose an 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 useful for all applications (cf., SQL–92's Transaction Isolation Levels [10, 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 timestamps 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. 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.

6 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 wide range of software vendors that would like to provide more advanced database functionality than offered by current products.

With specific design goals in mind, we explored what we believe to be central issues in the layered implementation of temporal functionality on a relational SQL–92 platform. We considered the options for the domain of timestamps, and for representing the temporal database variable *NOW*. 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. Finally, we considered the processing of temporal transactions.

This work points to several directions for future research. A more comprehensive study of the performance characteristics of layered implementation of temporal functionality is warranted. Next, issues related to the use of a control component in the layer should be explored. Finally, 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 while also reusing the services of an SQL–92 DBMS.

Acknowledgements

This research was supported in part by the Danish Natural Science Research Council, through grant 9400911, and the CHOROCHRONOS project, funded by the European Commission DG XII Science, Research and Development, as a Networks Activity of the Training and Mobility of Researchers Programme, contract no. FMRX-CT96-0056.

References

[1] I. Ahn and R. Snodgrass. Performance Analysis of Temporal Queries. *Inf. Syst.*, 49:103–146, 1989.

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

[3] M. Böhlen and C. S. Jensen. Seamless Integration of Time into SQL. TR-96–2049, Department of Computer Science, Aalborg University, Dec. 1996.

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

[5] J. Clifford and A. Tuzhilin, editors. *Recent Advances in Temporal Databases*. Workshops in Computing Series, Springer-Verlag, Nov. 1995.

[6] C. Davies, B. Lazell, M. Hughes, and L. Cooper. Time is Just Another Attribute—or at Least, Just Another Dimension, pp. 175–193. In [5].

[7] M. Finger and P. McBrien. On the Semantics of 'Current-Time' In Temporal Databases. *11th Brazilian Symposium on Databases*, pp. 324–337, Oct. 1996.

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

[9] T. Y. C. Leung and H. Pirahesh. Querying Historical Data in IBM DB2 C/S DBMS Using Recursive SQL, pp. 315–331. In [5].

[10] J. Melton and A. R. Simon. *Understanding the New SQL: A Complete Guide*. Morgan Kaufmann Publishers, 1993.

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

[12] B. Salzberg. Timestamping After Commit. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, pp. 160–167, Sep. 1994.

[13] A. R. Simon. *Strategic Database Technology: Management for the Year 2000*. Morgan Kaufmann Publishers, 1995.

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

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

[16] R. T. Snodgrass. A Road Map of Additions to SQL/Temporal. ANSI, Feb. 1996.

[17] R. T. Snodgrass, editor. *The TSQL2 Temporal Query Language*. Kluwer Academic Publishers, 1995.

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

[19] A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, editors. *Temporal Databases: Theory, Design, and Implementation*. Benjamin/Cummings Publishers, 1993.

[20] C. Vassilakis, N. Lorentzos, and P. Georgiadis. Transaction Support in a Temporal DBMS, pages 255–271. In [5].

[21] S. B. Zdonik and D. Maier, editors. *Readings in Object-Oriented Database Systems*. Morgan Kaufmann Publishers, 1990.