40

Effective Timestamping in Databases

Kristian Torp, Christian S. Jensen, and Richard T. Snodgrass

Many existing database applications place various timestamps on their data, rendering temporal values such as dates and times prevalent in database tables. During the past two decades, several dozen temporal data models have appeared, all with timestamps being integral components. The models have used timestamps for encoding two specific temporal aspects of database facts, namely transaction time, when the facts are current in the database, and valid time, when the facts are true in the modeled reality. However, with few exceptions, the assignment of timestamp values has been considered only in the context of individual modification statements.

This paper takes the next logical step: It considers the use of timestamping for capturing transaction and valid time in the context of transactions. The paper initially identifies and analyzes several problems with straightforward timestamping, then proceeds to propose a variety of techniques aimed at solving these problems. Timestamping the results of a transaction with the commit time of the transaction is a promising approach. The paper studies how this timestamping may be done using a spectrum of techniques. While many database facts are valid until *now*, the current time, this value is absent from the existing temporal types. Techniques that address this problem using different substitute values are presented. Using a stratum architecture, the performance of the different proposed techniques are studied. Although querying and modifying time-varying data is accompanied by a number of subtle problems, we present a comprehensive approach that provides application programmers with simple, consistent, and efficient support for modifying bitemporal databases in the context of user transactions.

1 Introduction

In a wide range of database applications, accountability and traceability are important; such applications manage transaction-time databases, where all previous database states are retained. In addition, many database applications require the times when the facts stored in the database are true to be stored with these facts. Such applications manage valid-time databases. A database recording both transaction and valid time is a bitemporal database [13].

The goal of this paper is to provide an effective approach to timestamping of data that may be used directly by application developers as well as may be employed within a stratum, which is a layer on top of a database management system (DBMS) that translates statements in a temporal query language into conventional SQL. The stratum should ensure ACID properties on the user transactions, by exploiting the transaction and concurrency control facilities of the underlying DBMS, specifically SQL's COMMIT and ROLLBACK statements. In particular, we do not allow any modifications to the DBMS itself, rendering the approach relevant also to non-DBMS vendors.

Designing a mapping of user transactions to SQL that provides the desired semantics turns out to be a challenging task. We shall see that oft-proposed approach of using the commit time of a transaction as the timestamp value of its database modifications is difficult to realize in practice, especially from the outside of the DBMS. One problem is that the commit time only becomes known when a transaction has exhausted all its statements, and so the commit time cannot be used in those statements. Consequently, a (single, temporary) transaction-internal transaction time, no later than the time of the first modification statement, must be used in order to make the results of modification statements visible within the transaction itself. This raises the concern of what value should be used for this temporary time value, and how and when it should be replaced with the permanent value.

This paper analyzes the implications of supporting transaction time in the presence of transactions and in the context of a stratum architecture. The paper proposes and studies the properties, including performance, of a range of techniques for updating database records resulting from a transaction's modifications to reflect the permanent commit time that only becomes available at commit time.

As a next step the paper also considers valid time, whose characteristics differ from those of transaction time. Valid times are user-specified or given by the system using default values—transaction times are always system-specified. The user may use the variable time value *now* that denotes the current time for delimiting valid time periods. Also unlike transaction time, valid times cover the entire time domain from "beginning" to "forever"—transaction-time values never exceed the current time.

The paper shows that when the systems assigns default valid-time values, valid time must be handled identically to transaction time. Otherwise, the user can extract database states that are inconsistent. When the `CURRENT_DATE` function (as well as the associated `CURRENT_TIME` and `CURRENT_TIMESTAMP` functions [15]) is present, we show that the value returned must be the commit time of the transaction. It turns out that the use of the commit time may lead to (illegal) periods that start after they end. When this occurs, the intermediate result of a modification, computed during the execution of the transaction, is different from the final result, computed at the commit time of the transaction. While this phenomenon cannot be eliminated, we show that it can be detected and subsequently handled via transaction abortion.

The performance of timestamping during modifications is a major concern. A performance study shows that the solutions suggested in the paper have efficient implementations, both for applications handling time-varying data explicitly in applications and in temporal databases handling time-varying data implicitly.

We conclude that although querying and modifying time-varying data is accompanied by a number of problems of surprising subtlety, it is possible to provide application programmers with simple, consistent, and efficient support for bitemporal databases in the context of user transactions, without requiring any changes to the underlying DBMS.

The paper is organized as follows. The next section introduces the stratum approach to implementing temporal databases. Requirements for correctly supporting transactions handling time-varying data are listed in Section 3. We then outline a new approach in Section 4. Sections 5 and 6 provide the details for effecting correct timestamping of the transaction-time and the valid-time dimensions, respectively. Different approaches for timestamping both valid time and transaction time are compared in Section 7. A performance study of design alternatives is presented in Section 8. Related work is discussed in Section 9, and Section 10 summarizes and points to directions for future research.

2 Temporal Databases and Stratum Architecture

As a first step in introducing the topic of the paper, we briefly describe bitemporal data. This type of data has associated a valid time, indicating when the data was true in the modeled reality, and a transaction time, indicating when the data along with its valid time was stored as current in the database.

The valid time of a tuple, a period, may be recorded using the two attributes `V-Begin` and `V-End`, and similarly, the transaction-time period of a tuple, also a period, may be recorded using attributes `T-Start` and `T-Stop`. We use half-open time periods.

A sample bitemporal table is shown in Table 1. The first tuple was recorded in the database on January 1, 1998, stating that Joe was with the Shoe department from that day onward. The variables *nobind now* and *until changed* will be explained in detail shortly; for now, assume they both mean "until we learn more." The three next tuples record that Bob is with the Outdoor department in the period [1998-01-04 - 1998-01-11), Jim is with the Toy Department in the period [1998-01-04 - 1998-01-12), and Jill is with the Shoe department in the period [1998-01-14 - 1998-01-19). This was all recorded on the 2^{nd} of January. The information regarding Joe was believed correct until January 20, when it was discovered that Joe was only in the Shoe department until January 8, at which time he had been transferred to the Toy department. As a result, the initial information was logically deleted, by placing 1998-01-20 in the `T-Stop` attribute of the first tuple, and by inserting the second tuple. A tuple (the third) was inserted during that same transaction, on January 20, to reflect that Joe had been in the Toy department since January 8. Finally, on January 23, we learned that this was incorrect: in reality Joe had been transferred to the Outdoor department, rather than the Toy department, on January 8. This led us to logically delete the third tuple for Joe and insert the final tuple. As can be seen, the benefit of a bitemporal table is that it captures the history of the enterprise, as well as the sequence of changes to that history.

`Name`	`Dept`	`V-Begin`	`V-End`	`T-Start`	`T-Stop`
Joe	Shoe	1998-01-01	*nobind now*	1998-01-01	1998-01-20
Bob	Outdoor	1998-01-04	1998-01-11	1998-01-02	*until changed*
Jim	Toy	1998-01-04	1998-01-12	1998-01-02	*until changed*
Jill	Shoe	1998-01-14	1998-01-19	1998-01-02	*until changed*
Joe	Shoe	1998-01-01	1998-01-08	1998-01-20	*until changed*
Joe	Toy	1998-01-08	*nobind now*	1998-01-20	1998-01-23
Joe	Outdoor	1998-01-08	*nobind now*	1998-01-23	*until changed*

Table 1: The Bitemporal Table, `Emp`

A number of quite different and more or less temporally enhanced query languages exist that permit an application programmer to modify and query bitemporal tables [29]. For example, SQL-92 [15] and SQL3 provide little built-in support, leaving more work to the application programmer. Other languages such as TSQL2 [21] and ATSQL [3] extend SQL-92 and provide advanced support, making application development easier.

Using an integrated DBMS architecture to implement a temporal data model that extends SQL with temporal support is a costly task, which only the major DBMS vendors can accomplish. The fact that existing DBMSs already manage large quantities of temporal data suggests that a better approach is available: providing built-in temporal support to applications by interposing a stratum between an existing DBMS and the application.

The stratum exploits the services already provided by the DBMS to offer temporal support to the application. Indeed, to be cost-effective, this approach is used by some vendors to enhance their own systems [25, 26]. By adopting a stratum approach, it is possible to maximally reuse existing technology and relatively quickly make a temporal DBMS available to the application programmers so they will benefit from the built-in temporal support of a temporal query language. Among the disadvantages of using a stratum approach is the inapplicability of well-known temporal storage structures, temporal indices, and algorithms that implement temporal operations such as temporal join, coalescing, and timeslicing algorithms.

In this paper, we assume a stratum architecture and thus aim to reuse the services provided by an existing DBMS, which is itself considered a black box. The consequence is that the techniques and results presented here are relevant for the layered implementation of a temporal DBMS, as well as for application programmers who do not have built-in temporal support available, but must handle the temporal aspects directly in their applications. The stratum architecture is illustrated in Figure 1, where the downward arrows denote flows of queries, the upward arrows denote flows of data, and the boxes are software components.

In this figure, the user first enters a temporal statement. The stratum converts the temporal statement to an SQL-92 statement that is executed in the underlying conventional DBMS. The DBMS sends the result back to the stratum, which then displays the result of the statement to the user. The user cannot see that the data is actually stored in a conventional DBMS—the stratum encapsulates the DBMS from the user's point of view.

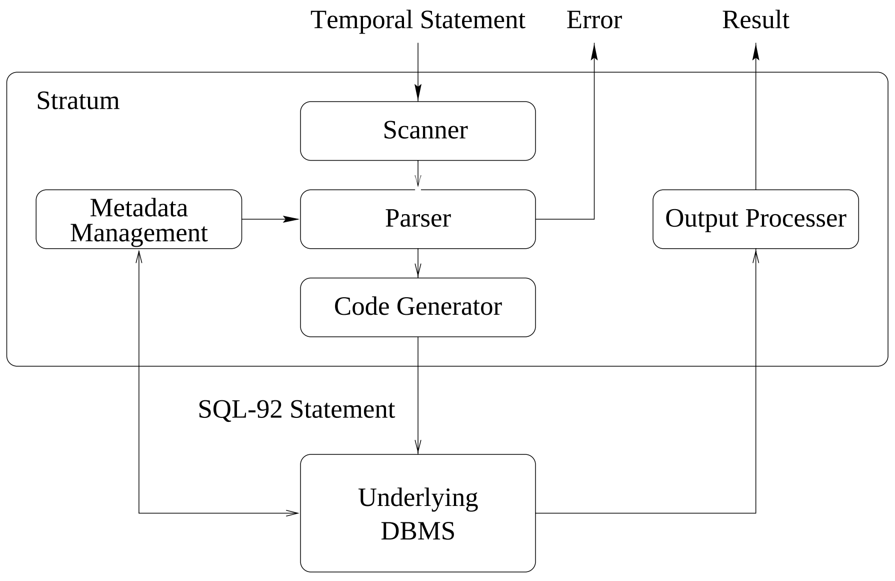


Figure 1: The Stratum Architecture

The stratum approach is similar in some ways to the related area of *mediators* [27, 28] and, more generally, of *integration architectures*. Broadly speaking, a mediator offers a consistent data model and accessing mechanism to a range of disparate data sources. The two approaches share an emphasis on interposing a layer, termed the mediator (also called a *wrapper* [18]) that changes the data model of the data, or allows new query facilities for accessing the data. A stratum differs from a mediator in that it is fully cognizant of the particular characteristics of the underlying DBMS and can exploit the constructs and facilities that the DBMS provides.

In the approach for timestamping advocated here, only few assumptions need be made about the temporal data model implemented using the stratum architecture. In particular, the specifics of the built-in support for querying are not important; only the facilities for database modification are of interest. So we now describe precisely how temporal modification statements are translated into SQL-92 modification statements.

We examine two simple types of modification statements. The first comprises the ones allowed in SQL: `INSERT`, `DELETE`, and `UPDATE`. Here the syntax is exactly that specified in SQL, with the stratum automatically supplying the valid-time and transaction-time timestamps, consistent with the semantics expected of these timestamps. As an example, the following is a valid SQL-92 statement.

```
UPDATE Emp
SET Dept = 'Toy'
WHERE Name = 'Joe'
```

As `Emp` is a bitemporal table, the semantics of this statement, consistent with the snapshot semantics of SQL-92, is to change Joe's department now, and in the future.

For queries, we use the traditional SQL-92 `SELECT` statement, perhaps with explicit reference to the timestamp attributes.

Table 2 shows how the modification statements may be mapped to SQL-92. The left column gives temporal query language statements for insertion and deletion (updates are combinations of deletions and insertions). Here we use the syntax proposed for SQL3 [22], though we emphasize that the specific syntax is not important. The right column provides the translation to SQL-92 effected by the stratum and thus defines the semantics of the temporal statements. We elaborate on each translation below. The translation is preliminary because the representations of *now*, *nobind now*, *until changed*, *start value*, and *stop value* using values of SQL-92 data types are not specified. Later sections will study the issues involved in providing such values, resulting in a fully specified definition of the modification statements.

When we insert a tuple (the mapping for such an insertion appears as the second row of Table 2), it is timestamped with the period `[`*now* `-` *nobind now*`)` in the valid-time dimension. This states that the fact is valid from the current time until we learn more. In the transaction-time dimension, it is timestamped with the

Temporal statement:

`CREATE TABLE Emp (Name VARCHAR (20) Dept VARCHAR (20))`
`AS VALIDTIME PERIOD(DATE) AND TRANSACTIONTIME`

Corresponding SQL-92 statement:

`CREATE TABLE Emp ( Name VARCHAR (20), Dept VARCHAR (20),`
`V-Begin DATE, V-End DATE, T-Start DATE, T-End DATE)`

Temporal statement:

`INSERT INTO Emp VALUES (`*new name*`,` *new dept*`)`

Corresponding SQL-92 statement:

`INSERT INTO Emp VALUES (`*new name*`,` *new dept*`,` *now*`,` *nobind now*`,`
start value`,` *until changed*`)`

Temporal statement:

`VALIDTIME PERIOD [`*Start* `-` *Stop*`)`
`INSERT INTO Emp VALUES (`*new name*`,` *new dept*`)`

Corresponding SQL-92 statement:

`INSERT INTO Emp VALUES (`*new name*`,` *new dept*`,` *Start*`,` *Stop*`,`
start value`,` *until changed*`)`

Temporal statement:

`DELETE FROM Emp WHERE` *Predicate*

Corresponding SQL-92 statement:

`INSERT INTO Emp`
`SELECT Name, Dept, V-Begin,` *now*`,` *start value*`,` *until changed*
`FROM Emp WHERE` *Predicate* `AND T-Stop =` *until changed*
`AND V-Begin <` *now* `AND` *now* `< V-End;`
`UPDATE Emp SET T-Stop =` *stop value*
`WHERE` *Predicate* `AND T-Stop =` *until changed*
`AND V-Begin <` *now* `AND` *now* `< V-End`

Table 2: Initial Translation of Temporal Modification Statements

period `[`*start value* `-` *until changed*`)`, denoting that it was present in the database starting at *start value* and persists to now, that is, until a future transaction, or a future statement in the current transaction deletes or updates the tuple.

A deletion of a tuple (the fourth row of Table 2) is effected by updating the `T-Stop` attribute to the *stop value*, indicating that our old belief no longer holds, and inserting a tuple to record our new belief that the tuple was valid in the modeled reality from the old `V-Begin` time to the current time (*now*). Note that all explicit attributes are copied. Because the insertion uses a `SELECT`, it must appear before the update statement. Note also that the inserted tuple will not be later changed by the update statement, because *now* `< V-End` will not hold for the inserted tuple.

Not shown in the above table is the translation of an SQL-92 update statement (with an implicit valid-time period of "now" to "forever"). Such an update can be stated as a temporal deletion of the old values, coupled with a temporal insertion of the new values.

3 Correct Transactions

This section concerns the correctness of transactions. We first review the notion of correct transactions in snapshot databases. Next, we turn to discuss the correctness of temporal transactions and illustrate several subtle problems that arise when the correctness criteria of transactions on snapshot databases are generalized to temporal databases. The discussion of correct temporal transactions in this section is independent of implementation techniques, e.g., for concurrency control, recovery, and temporal attribute visibility.

3.1 Correct Snapshot Transactions

We define snapshot transactions and temporal transactions as database transactions on snapshot tables and temporal tables, respectively. Note that we do not differentiate time values stored in explicit attributes (handled in an ad-hoc fashion by applications) from time values stored in implicit attributes (handled by a temporal DBMS).

The correctness criteria for snapshot transactions running at isolation level SERIALIZABLE [15] are the ACID properties [12]. These properties, guaranteed by the DBMS, state that the transaction is an atomic unit of execution: it commits or it aborts in its entirety. After the transaction has either committed or aborted, the database will be in a consistent state according to, e.g., primary key constraints, referential integrity constraints, and `CHECK` statements. The execution of a transaction is isolated from the execution of other transactions. Finally, the database-state changes caused by the transaction are made durable.

3.2 Correct Temporal Transactions

In the transition from snapshot transactions to temporal transactions, the novel aspect is that we apply special semantics to the timestamp attributes. The three differences between snapshot transactions and temporal transactions are as follows. (1) For snapshot transactions we store modifications made to tuples; for temporal transactions we store in addition when the modifications took place. (2) The time when tuples were modified can be queried in a temporal transaction. (3) In temporal transactions, the semantics of `CURRENT_DATE` must be consistent with the timestamps stored in the database.

For temporal transactions to be upwards compatible [1] with snapshot transactions and because temporal transactions are not fundamentally different from snapshot transactions, we want to retain the ACID properties as correctness criteria. In particular, we wish to retain the view that transactions logically have no duration (i.e., that they appear to execute instantaneously) and that this execution corresponds to a serial execution in commit order. The transaction timestamp should be consistent both with the commit order and with the clock time when the transaction committed.

One might assume that as a DBMS has the necessary mechanisms for providing the ACID properties for snapshot transactions, the DBMS will automatically also retain the ACID properties for temporal transactions. However, we will show that the timestamp attributes have to be handled carefully to avoid violating the ACID properties.

Problems Occuring in Temporal Transactions

To motivate the need for additional requirements to temporal transactions, we illustrate the new problems that may occur in temporal transactions by Table 1 and the example in Figure 2. For convenience, all timestamps on the figure are dates during the month of January, 1998, and we make the transactions artificially long to emphasize the semantic problems that may occur. Note that the problems we illustrate may occur in DBMSs using two-phase locking.

Although our emphasis is on database modification statements, we occasionally need the ability to observe the contents of the database being modified. For this purpose, we use the SQL-92 query given below, denoted $Q(t)$, that retrieves the snapshot state of Table 1 (the `Emp` table) as of a time instant t.

```
SELECT Name, Dept
FROM   Emp
WHERE  V-Begin  <= t AND t < V-End  AND
       T-Start  <= t AND t < T-Stop
```

Above the time-line in Figure 2, we show the contents of the `Emp` table on day 4 for Bob and Jim. Bob is with the Outdoor department in the period [1998-01-04 - 1998-01-11) and Jim is with the Toy department in the period [1998-01-04 - 1998-01-12). Below the time-line we have shown two transactions, T_1 and T_2. On days 6 and 10, T_1 updates Bob to be with the Toy department and updates Jim to be with the Outdoor department, respectively. Further, T_1 retrieves the current state of the database, using $Q(t)$ on day 11. On day 8, T_2 updates Jim to be with the Sports department. T_1 starts on day 4 and commits on day 12; T_2 starts on day 7 and commits two days later. Hence, T_2 starts after T_1 starts and commits before T_1 commits.

First consider only transaction T_1 in Figure 2 and assume that the statements are evaluated using the translations outlined in Table 2 and the obvious approach of using `CURRENT_DATE` for the variables *now* and *start value* and the maximum

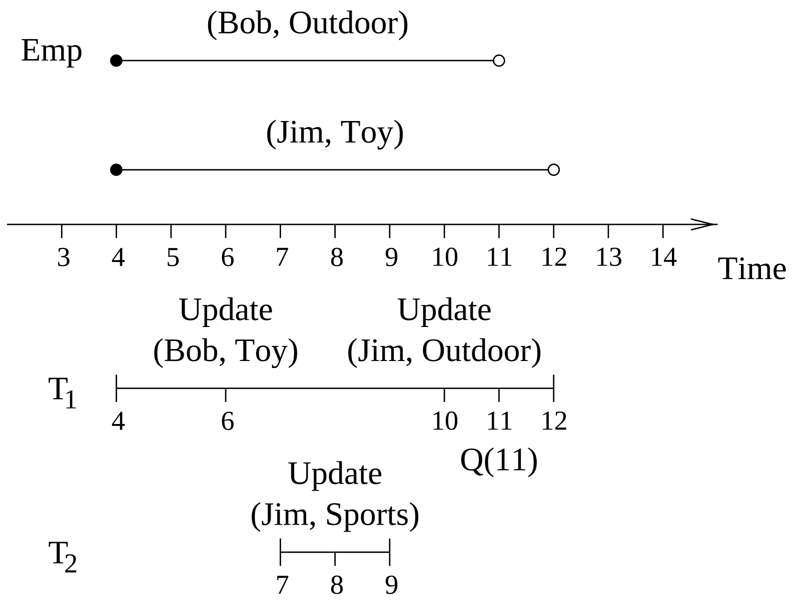


Figure 2: Problems Occuring in Temporal Transactions

value of the time domain for *nobind now* and *until changed* (in SQL-92, this is 9999-12-31). The following two problems may occur.

The first problem is that a query may see that the actual executions of transactions have durations in time. This makes it possible to perform a timeslice to an intra-transaction state obtaining a result that may violate the consistency requirements of the database. As an example, Bob and Jim exchanged departments during T_1; however, when we are using `CURRENT_DATE` for *now*, $Q(8)$ executed in a separate transaction at day 14 is able to detect that Bob and Jim are with the same department. This is due to (Bob, Toy) having a `V-Begin` of January 7 and (Jim, Toy) having a `V-End` of January 10, clearly missing our goal of transactions appearing to execute instantaneously.

The second problem is that a query, executed twice in a transaction and with no intermediate modifications, may return different results. Returning different results is similar to a non-repeatable read [12], which violates the isolation of transactions. As an example using Table 1, the execution of $Q($`CURRENT_DATE`$)$ in a separate transaction on day 13 would not include Jill in its result set; however the execution of $Q($`CURRENT_DATE`$)$ on day 15 *in the same transaction* will return Jill in its result set, because Jill is recorded being in the Sports department in the period `[`1998-01-14 `-` 1998-01-19`)`. Again, this breaks the logical illusion of the transaction being instantaneous.

The two problems mentioned above can be solved by using a single fixed value for *now*, *start value*, and `CURRENT_DATE` within a transaction. Assume we use the start times of the transactions and consider transactions T_1 and T_2 in Figure 2. While our new approach solves the first two problems, it also introduces two new problems.

The first problem is that the outcome of temporal transactions is inconsistent with the outcome for equivalent SQL-92 transactions. This violates temporal upwards compatibility [1], which requires that nontemporal transactions and queries should return the same results when applied to tables with temporal support as when applied to non-temporal tables. As an example, $Q(11)$ in T_1 will return the two tuples (Bob, Toy) and (Jim, Sports), whereas the equivalent (nontemporal) SQL-92 transaction will result in (Bob, Toy) and (Jim, Outdoors). To understand why this occurs, look at the content of the `Emp` table when T_1 is ready to make the update at day 10. The content of the `Emp` table for Bob and Jim is shown in Table 3. In this table, only two current tuples are currently valid: (Bob, Toy) (tuple 4) and (Jim, Sports) (tuple 6).

`Name`	`Dept`	`V-Begin`	`V-End`	`T-Start`	`T-Stop`
Bob	Outdoor	1998-01-04	1998-01-11	1998-01-01	1998-01-04
Jim	Toy	1998-01-04	1998-01-12	1998-01-02	1998-01-07
Bob	Outdoor	1998-01-04	1998-01-04	1998-01-04	*until changed*
Bob	Toy	1998-01-04	*nobind now*	1998-01-04	*until changed*
Jim	Toy	1998-01-04	1998-01-07	1998-01-02	1998-01-07
Jim	Sports	1998-01-07	*nobind now*	1998-01-07	*until changed*

Table 3: Part of Table `Emp` on Day 10 before the Update of Jim in T_1

The update at day 10 in transaction T_1 should update the last tuple in Table 3, because this is the tuple with current and currently valid information for Jim. However, we are using the start time of a transaction for *now* in the transaction, which is day 4 and day 7 for transactions T_1 and T_2, respectively. Because transaction T_2 is executed during T_1, the predicate for a translated delete (the fourth tuple in Table 2) will fail on the last tuple in Table 3 because day 4 is not between the values of `V-Begin` (day 7) and `V-End` (*nobind now*). Recall that an update is a combination of a delete and an insert.

The second problem using the start time for *now* is what may be termed history correction, which undermines the accountability of a temporal database. As an example the execution of $Q(5)$ at day 5 in a separate transaction will return (Bob, Outdoor) and (Jim, Toy), because T_1 had not yet committed. However, the execution of $Q(5)$ at day 14 in another separate transaction will return (Bob, Toy) and (Jim, Toy).

To solve the two problems using the start-time of transactions for *now*, we can extend the `WHERE` clauses for the delete in the fourth row of Table 2 to include a check of whether the `V-End` attribute is equal to *nobind now*. The predicate for the valid-time dimension is thus extended from "`V-Begin` < *now* `AND` *now* < `V-End`" to "`V-Begin` < *now* `AND (`*now* < `V-End OR V-End` = *nobind now*`)`". All prior

SQL-92 statements of the transaction are guaranteed not affect tuples with `V-End` = *nobind now*. With this extension, the outcome of T_1 and T_2 is as shown in Table 4.

`Name`	`Dept`	`V-Begin`	`V-End`	`T-Start`	`T-Stop`
Bob	Outdoor	1998-01-04	1998-01-11	1998-01-01	1998-01-04
Jim	Toy	1998-01-04	1998-01-12	1998-01-02	1998-01-07
Bob	Outdoor	1998-01-04	1998-01-04	1998-01-04	*until changed*
Bob	Toy	1998-01-04	*nobind now*	1998-01-04	*until changed*
Jim	Toy	1998-01-04	1998-01-07	1998-01-02	1998-01-07
Jim	Sports	1998-01-07	*nobind now*	1998-01-07	1998-01-04
Jim	Sports	1998-01-07	1998-01-04	1998-01-04	*until changed*
Jim	Outdoor	1998-01-04	*nobind now*	1998-01-04	*until changed*

Table 4: Part of Table `Emp` after the Execution of T_1

Query $Q(11)$ in transaction T_1 now returns the same result as the equivalent SQL-92 transaction. Further, we no longer see Bob and Jim in the same department when executing $Q(5)$ at day 14 in a separate transaction. However, this approach introduces yet another problem: the start time of a period recorded in a temporal table can be after the stop time. This is obviously a violation of the properties of a period. The new problem is shown in the sixth and seventh tupls in Table 4 that shows the result after executing T_1 and T_2 from Figure 2 with the predicate changed for the valid-time dimension.

The fundamental problem of using the start time of a transaction for *now* is retroactive modifications of exposed tuples. That is, a transaction with start time t_a can modify a tuple also modified by a transaction with start time t_b, and $t_b > t_a$. The problem was illustrated by transactions T_1 and T_2 in Figure 2. These both update the tuple for Jim. T_1 starts at day 4 and T_2 starts at day 7. However, T_2 modifies and commits before T_1 modifies Jim, causing the problems described above.

To avoid the problems associated with using the start time for *now*, we will finally use the commit times of the transactions as their values for *now* and for `CURRENT_DATE`. When the value of either is needed, we postpone applying the value until the transaction is ready to commit. This approach solves all of the above problems, but again introduces a new problem. We cannot return a result of $Q(11)$ in transaction T_1 because the we do not know the value for *now* for the tuples modified during the transaction until the transaction actually commits at time 12.

The central question is, what the semantics should be for the transactions shown in Figure 2 and indeed, for any transaction applied to a temporal database. Once the appropriate semantics has been determined, one can then consider how to implement that semantics in a stratum architecture.

Requirements to Temporal Transactions

Starting with the ACID properties and considering the problems illustrated in the previous section, we now enumerate a set of requirements for a consistent, logical semantics for temporal transactions.

Requirement 1 All of the tuples modified by a single transaction must be given the same timestamp value in the database. Otherwise, we have shown the problem of being able to timeslice and see intra-transaction states, thus violating the consistency of the database.

Requirement 2 The value of `CURRENT_DATE` must be fixed within a transaction. If `CURRENT_DATE` changes in a single transaction, we have shown that the query $Q($`CURRENT_DATE`$)$ on Table 1 on days 13 and 15 in the same transaction return different results. Fixing the value of `CURRENT_DATE` in a transaction is a refinement of the SQL-92 semantics, in which the value of `CURRENT_DATE` is fixed only within a statement, but may change within a transaction[1].

Requirement 3 The start time of a time period assigned to a tuple must be smaller than or equal to the stop time of the period. Otherwise, we violate the properties of a period.

Requirement 4 The timestamp used for a transaction should not be after the commit time of the transaction. Using a timestamp after the commit time, e.g., by adding two days to the start time of the transaction and aborting all transactions running for more than two days, results in an inaccuracy where a tuple is not visible for query $Q($`CURRENT_DATE`$)$ from the time the transaction actually commits until the time chosen as the commit time of the transaction.

Requirement 5 The result of $Q(t_1)$ at time t_2 should the same as the result of $Q(t_1)$ at time t_3 where $t_1 \leq t_2$ and $t_1 \leq t_3$. As discussed above, $Q(5)$ on day 5 in a separate transaction will return Bob as being with the Outdoor department. However, if we use the start time for timestamping, $Q(5)$ on day 14 in another transaction will return Bob as being in the Toy department.

[1]While the value is fixed within a statement, *which* fixed value to use is left entirely to the implementor. General Rule 3 of Subclause 6.8 <datetime value function> of the SQL-92 standard states “If an SQL-statement generally contains more than one reference to one or more <datetime value function>s, then all such references are effectively evaluated simultaneously. The time of evaluation of the <datetime value function> during the execution of the SQL-statement is implementation-dependent.” [14, p. 110].

Requirement 6 Temporal transactions must be able to see their own modifications, e.g., after a transaction has updated a tuple, all explicit attributes of the updated tuple must be visible to a query immediately following the update in the transaction.

Requirement 7 A transaction must be able to see the values of timestamp attributes of tuples it had previously modified.

Requirement 8 The level of concurrency in a database should not be lowered significantly when temporal support is added, such as by requiring that all transactions be executed sequentially.

Requirement 9 The timestamping approach should not restrict the temporal query language. As an example, a very efficient timestamping approach that disallows valid-time periods into the future is not appealing. Such a restriction would reduce the benefits of a temporal database.

These requirements are used as guidelines for designing a timestamping approach for transaction-time and valid-time tables. Unfortunately, some of these requirements are mutually exclusive or may affect each other. We discuss this in the following, but discuss first the implications of assuming a stratum architecture.

We want to implement the timestamping approach using a stratum architecture. The rationale for building on top of a conventional DBMS is to be able to reuse its functionality. Most of the major DBMSs use locking as a concurrency control mechanism [4]. We therefore assume that two-phase locking is used to provide the isolation property of the ACID properties. The use of two-phase locking has some impacts on timestamping in the stratum, as explained next.

Salzberg has shown that to achieve a transaction-consistent view of previous database states (Requirement **R1**), it is necessary to use the same timestamp for all modifications within a transaction [19]. The timestamp must be after the time at which all locks have been acquired. Otherwise, the timestamps will not properly reflect the serialization order of transactions [19].

To make it possible for transactions to see their own modifications (Requirement **R6**), it may be necessary to associate timestamps with tuples before all locks have been acquired [19]. Specifically, at the time of the first modification in a transaction, we may not have all locks, but we must associate a timestamp with the modified tuples because a query follows the modification. However, it is not possible to get the permanent timestamp.

The two requirements to temporal transactions discussed above, retaining a transaction-consistent view of previous database states (Requirement **R1**) and that transactions should be able to see their own modifications (Requirement **R6**), can be fulfilled by using *timestamping after commit* [19]. However, timestamping after commit does not make it possible for a temporal transaction to see the values of timestamp attributes it has previously modified (Requirement **R7**). Hence, Requirements **R1** and **R7** are mutual exclusive. The first requires that we use a value for *now* after all locks are acquired, whereas the second requires that we use a value before the first lock is acquired.

Retaining transaction-consistent previous database states (Requirement **R1**) is more important than seeing permanent timestamps within transactions (Requirement **R7**). We therefore focus on how to make previous states transaction-consistent. The unavoidable consequence is that the value of `T-Start` or `T-Stop` for tuples modified by a transaction are not known during the transaction.

The use of the commit time for timestamping modifications is dictated by Requirement **R1** and affects Requirement **R9**. As an example, had the query $Q(11)$ in transaction T_1 in Figure 2 selected the `T-Start` and `T-Stop` attributes, this would have to result in an error or a warning. Minimizing the effect on the temporal query language is discussed in Section 4.

Note that the use of the commit time for *now* does not result in significantly lowering the level of concurrency in a database (Requirement **R8**). However, locks must be held slightly longer; this is examined in Section 8.5.

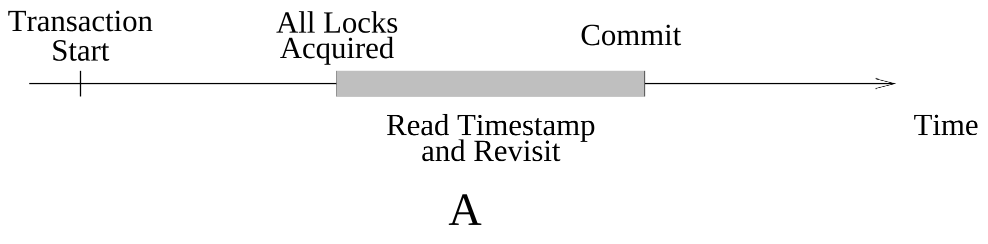
4 A New Approach

We now show how the problem of not knowing the permanent timestamp of modified tuples within a transaction can be minimized, and, when it occurs, how it can be easily detected. Specifically, we propose using a temporary value for the timestamp and then revisit tuples after all locks have been acquired, to replace the temporary value with the (now-known) permanent timestamp.

There are three major differences between the approach presented here and previous approach to timestamping [19, 23]. First, we consider both valid-time and transaction-time compared to transaction-time only. Second, we consider an entire temporal query language and do not restrict ourselves to timeslice (“as-of”) queries. In particular, we consider the display of the temporal attributes. Third, we assume a stratum architecture, meaning that we reuse the services of, but also cannot change, an underlying DBMS.

Figure 3 illustrates our approach. In Figure 3A, we show the times when a transaction starts, when it has acquired all locks, and when it commits. The shaded strip indicates the time period, from the time when all locks have been acquired to the time when the transaction commits, where it is possible to revisit and update tuples with their permanent timestamp. Tuples modified between the time the transaction started and the time when all locks were acquired must be revisited.

In a conventional DBMS, typically using strict two phase locking [2], it is not known that all locks have been acquired until when the transaction's final statement is reached, i.e., at user-commit. Further, a stratum has no access to the internals of the underlying DBMS. We therefore postpone reading the timestamp until after user-commit and then revisit the tuples modified by the transaction, to apply the permanent timestamp; the transaction then actually commits by having the stratum issue a commit to the underlying DBMS. This sequence of events is illustrated in Figure 3B.



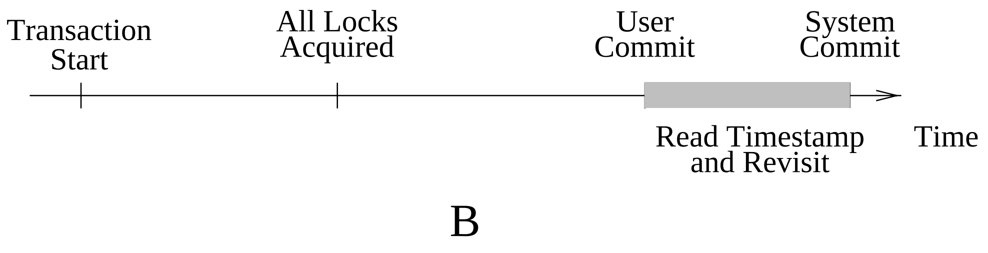


Figure 3: A: Temporal Transaction B: Mapping a Temporal Transaction to SQL-92

Using the approach shown in Figure 3B, we read the timestamp after user-commit, referred to as the *commit time*. Because we do not yet know the commit time when a transaction modifies a tuple, we set the appropriate transaction-time attribute to a temporary value and store in the tuple the transaction-id of the transaction modifying it. After user-commit, we read the system clock and save the transaction-id and the timestamp in a `CommitTime` table, which has the schema (`TID INTEGER, Commit-Time DATE`). We subsequently revisit all tuples modified by the transaction and apply the timestamp stored in the `CommitTime` table. We remove the transaction-ids from tuples and delete the entry in the `CommitTime` table. This extra step is similar in some ways to the revisit step in Postgres [23]. We term our approach *Timestamping after Commit with Revisitation*.

The use of timestamping after commit with revisitation adds two additional requirements to temporal transactions. First, timestamps that eventually are identical (i.e., are associated with tuples modified by the same transaction) must not appear temporarily to be different. Otherwise, the wrong result will be returned when timestamps are compared. Second, the effect of temporal modifications within a transaction should be easily understandable to the user writing the transaction.

The specifics of how to implement the timestamping after commit with revisitation in a stratum is discussed in detail in the next two sections. First, we discuss timestamping the transaction-time dimension, then examine the consequences of introducing also the valid-time dimension.

5 Transaction Timestamping

In this section we describe how timestamping in a transaction-time table can be achieved. Where there are obvious choices, we identify a particular approach. Where there are several possibilities, we list them and postpone choosing one approach until Section 7, where we present a specific approach to timestamping.

We first give an example that raises four design issues with respect to transaction timestamping. The issues are discussed in turn, thus providing the details of how to implement timestamping after commit for transaction time.

5.1 An Example

As an example of timestamping a transaction-time table, consider the `Emp` table from before that stores the names and departments of employees. To create `Emp` as a transaction-time table, we issue the temporal statement `CREATE TABLE Emp (Name VARCHAR(30), Dept VARCHAR(30)) AS TRANSACTIONTIME`[2]. As three separate transactions, we issue an insertion, an update, and an update and a query, as indicated in Figure 4. The modifications are expressed in plain SQL-92; the temporal semantics automatically supplies values for the tuple timestamps, as discussed informally in Section 2.

```
-- on 1998-01-06:
INSERT INTO Emp VALUES ('Joe', 'Shoe'); COMMIT;
-- on 1998-01-16:
UPDATE Emp SET Dept = 'Sports'
WHERE Name = 'Joe'; COMMIT;
-- on 1998-01-27:
UPDATE Emp SET Dept = 'Outdoor' WHERE Name = 'Joe';
SELECT Name, Dept, T-Start, T-Stop, FROM Emp; COMMIT;
```

Figure 4: Using the Transaction-Time Table `Emp`

Table 5 shows the `Emp` table after the three transactions commit. As can be seen from Table 5 we add two time attributes, `T-Start` and `T-Stop`. The time attributes are called *implicit attributes*, and `Name` and `Dept` are called *explicit*

[2]Again, the details of the temporal extensions are not important. We use a particular syntax [22] only to provide a specific example for expository purposes.

attributes. The implicit attributes capture the time evolution of the table. We emphasize that `Emp` is a transaction-time table, and hence captures the state stored in the database over time.

Name	Dept	T-Start	T-Stop
Joe	Shoe	1998-01-06	1998-01-16
Joe	Sports	1998-01-16	1998-01-27
Joe	Outdoor	1998-01-27	*until changed*

Table 5: The Transaction-Time Table, `Emp`

Table 6 shows how temporal statements are mapped to SQL-92 statements by the stratum. This table is a simplification of Table 2, considering only transaction time and utilizing timestamping after commit with revisitation. For simplicity, we assume that all explicit attributes occur in modification statements. When we insert a tuple, it is timestamped with the period `[`*temporary value* `-` *until changed*`)`. A deletion of a tuple is mapped to an update of the `T-Stop` attribute of the tuple to *temporary value*. A tuple qualifies for deletion if it satisfies *Predicate* and is current. An update, not shown in Table 6, is implemented as a temporal delete of the old tuple followed by a temporal insert of the new tuple. We do not show the mapping of insertions and deletions with user-supplied times because such statements are permitted only for tables with valid-time support.

```
Temporal statement: INSERT INTO Emp VALUES (new name, new dept)
  Corresponding SQL-92 statement:
    INSERT INTO Emp VALUES (new name, new dept,
                                      temporary value, until changed)
Temporal statement: DELETE FROM Emp WHERE Predicate
  Corresponding SQL-92 statement:
    UPDATE Emp SET T-Stop = temporary value
    WHERE Predicate AND T-Stop = until changed
Temporal statement: COMMIT
  Corresponding SQL-92 statements:
    c ← CURRENT_DATE;
    INSERT INTO CommitTime VALUES (transaction-id, c);
    UPDATE Emp SET T-Start = c WHERE tuple inserted by transaction-id;
    UPDATE Emp SET T-Stop = c WHERE tuple deleted by transaction-id;
    DELETE FROM CommitTime WHERE TID = transaction-id;
    COMMIT
```

Table 6: Mapping Statements on Transaction-Time Tables into Equivalent SQL-92 Statements

When a user commits, we record the transaction-id and `CURRENT_DATE` in the `CommitTime` table. All tuples modified by the transaction are then revisited. Tuples inserted by the transaction have the `T-Start` attribute updated to the commit time of the transaction. Similarly, tuples deleted by the transaction have their `T-Stop` attribute updated. For the two `UPDATE` statements, the `WHERE` clause is deliberately vague; how to identify tuples modified by a transaction is described in Section 8. After cleaning up the `CommitTime` table, the transaction actually commits. If the modification statements in Figure 4 are translated as indicated in Table 6, the result is Table 5.

Studying this example raises four questions, which we address in turn in the following sections.

- What is the *temporary value* of the transaction-time attributes for tuples modified within a transaction? As an example, the selection in Figure 4 is executed before the transaction is committed. The `T-Stop` attribute of the second tuple and the `T-Start` attribute of the third tuple of Table 5 will then have a temporary value. Which value should be displayed for these attributes?
- When a transaction commits, the modified tuples must be revisited. In a multiuser system, how do we guarantee that tuples are updated with the appropriate commit time during the revisit phase?
- How should *until changed* be represented, e.g., in the `SELECT` in Figure 4?
- Must modified tuples be revisited before the transaction actually commits?

5.2 Finding a Temporary Timestamp Value

If tuples are to be timestamped with the commit time, tuple modification must be deferred until the transaction commits [19], rendering it impossible for a transaction to see its own modifications. Timestamping tuples with a temporary value before the commit time makes it possible for a transaction to see its own modifications.

Permanent transaction timestamps are first applied after user commit. A potential problem therefore occurs when a transaction first modifies the database and then queries it, referring to the transaction timestamps. For example, this problem occurs in the last transaction in Figure 4. In general, many temporal queries may refer to the tuples' timestamps. There are several possible responses to this situation. (1) We can disallow queries that access the timestamps. (2) We can treat it as a semantic error when a transaction modifies a tuple and subsequently queries the transaction time of that tuple. (3) We can warn the user during query analysis when a statement referencing a transaction time attribute is encountered after a modification statement. (4) We can simply return the temporary value stored.

Disallowing references to timestamps restricts the query language, which we prefer not doing (cf. requirement **R9**). Simply returning the temporary value can be a great surprise to users and may lead to misunderstandings (if the temporary timestamps are relative to the smallest timestamp, the user would indeed be surprised that the tuple appeared to be inserted in 1 A.D.!). This leaves us with the choice of making it a semantic error or issuing a warning. We find the warning more appropriate because allowing reference to transaction time after modifications within the same transaction is then a decision made by the user, rather than by the system. The warning is of the form: the transaction times displayed may change after the transaction commits.

The temporary value must fulfill two requirements. First, it must make the tuple qualify for the current transaction-time state when the transaction-time attributes are referenced in a `WHERE` clause. Second, it must be a sensible value to return when the transaction-time attributes are used in a `SELECT` clause. The possible choices for the temporary value are as follows.

- Use the start time of the transaction.
- Use the time when the temporary value is first needed, e.g., the time of the first modification.
- Use multiple values within a transaction, e.g., `CURRENT_DATE`.

The first two alternatives will make the modified tuple qualify for the current state and are sensible values to display to the user, along with a warning that the values change when the transaction commits. We rule out using multiple values of two reasons. First, as discussed in Section 3, this can lead to non-repeatable reads when the same query is executed twice in a transaction, e.g., displaying the timestamps of a tuple inserted by the transaction. Second, it makes tuples temporarily have different timestamps values for timestamps that eventually get the same value, which we do not allow (cf. requirement **R1** and the discussion in Section 4).

5.3 Associating Transaction-ids With Tuples

We use a transaction-id when revisiting tuples to identify the tuples being modified. There are several ways to associate a transaction-id with tuples. First, we can store the transaction-ids directly in the tuples. In such an approach, the stratum adds an extra attribute when it passes a `CREATE TABLE` statement to the underlying DBMS. Storing the transactions-ids in the tuples can be done in two ways: in an extra attribute or encoded in the timestamp attributes themselves.

Using an extra attribute is straightforward: we simply store the transaction-id in this attribute; Postgres uses this approach [23]. In contrast, storing the transaction-id in a transaction-time attribute requires type conversion, because the domain of transaction-time attributes differs from the domain of transaction-ids (typically `TIMESTAMP` versus `INTEGER`). Collision between the encoded transaction-ids and actual timestamps can be avoided because transaction timestamps are larger than the time when the database was created. Thus the transaction-ids can be relative to the smallest timestamp (typically 0001-01-01): the first transaction-id is mapped to the smallest value in the time domain, the second transaction-id is mapped to the second smallest value in the time domain, and so on. When storing the transaction-ids in the transaction-time attributes, we must in addition store the temporary value for the tuple in an auxiliary data structure.

The choice of using an extra attribute versus converting transaction-ids in order to associate a transaction-id within tuples represents a space-time trade-off. The conversion may be useful, but is not very elegant in SQL-92 where the conversion between `INTEGER` and `TIMESTAMP` is via an `INTERVAL` [15]. This means we first have to convert a transaction-id to an `INTERVAL` and next have to add the interval to the smallest value in the time domain. This manipulation would be done in the first and second rows of Table 6 before the insert and update in the second column. A reverse, two-step expression is needed to decode a transaction-id again when identifying the tuples to update in the second column of the third row in Table 6.

As another alternative, we may store the transaction-ids separately from the tuples. Here, the stratum defines for each explicit table a new table that stores the tuple-id, the transaction-id, and the timestamp attributes affected by the modification. This approach has the consequence that two tables must be updated for each modification, compared with one table when storing the transaction-ids directly in the tuples.

5.4 Representation of *until changed*

Another and separate issue is the representation of *until changed*. All tuples not logically deleted are timestamped with *until changed* in the `T-Stop` attribute as shown in Table 6. The value for *until changed* cannot be between the time the database was created and the current time; using a value in the near future is also not a safe option. These representations are ambiguous because we eventually will not be able to distinguish *until changed* from the value with which it is represented. Even avoiding these possibilities, several values are still available for representing *until changed*. Specifically, three possible values remain.

- Any time before the database was created.
- The largest value in the domain (9999-12-31 in SQL-92).
- The value `NULL`.

Using a value before the database was created implies that the transaction-time stop value may be smaller than the transaction-time start value, which we do not allows (cf. requirement `R3`). The requirement can be fulfilled by using the largest value in the domain. The last alternative, using `NULL` for *until changed*, is also possible because the transaction-time stop cannot be `NULL`: we can thus "reuse" `NULL` without overloading it. Further, `NULL` often requires less space in a database than other timestamps.

5.5 Strategies for Revisiting Tuples

Yet another issue is when to update temporary timestamps to the permanent commit times, or to be specific, when to execute the two updates and the delete for the translated commit statement in Figure 6. We prefer flexibility in scheduling these database modifications.

Because revisiting tuples adds to the system load, to be discussed further in Section 8.5, we first identify which modifications and queries need permanent timestamps. Second, we explore different approaches for updating the temporary timestamps to the permanent values, the purpose being to find the most efficient approach.

In Section 5.1, we described a scenario where the temporary values of the transaction-time attributes are updated to the commit time right after user-commit. Examining which modifications and queries that need to know the permanent transaction timestamps, we see that no modifications and queries on the current state depend on the permanent transaction timestamps; the current states are the tuples with *until changed* in the `T-Stop` attribute. Only modifications and queries on previous states depend on the permanent transaction timestamps for their correct execution. As queries on previous states are often syntactically identifiable (e.g., [3, 22]), syntactic analysis can decide when permanent transaction timestamps are required for reasons of correctness of query processing. As an example, selecting all tuples (with their transaction-timestamps) can be expressed as `TRANSACTION-TIME SELECT * FROM Emp` in an SQL3 proposal [22]. Selecting the current transaction-time state (without transaction-timestamps) is expressed simply as `SELECT * FROM Emp`. The keyword `TRANSACTIONTIME` makes it possible to determine when correct, permanent timestamps are needed and not needed.

For modifications and queries not requiring the permanent timestamps, there are several approaches to the revisiting of tuples to apply the permanent timestamping.

- *Eager*: For each transaction, the permanent timestamp is applied immediately, at user-commit.
- *Low-system-usage*: On low system load, e.g., during lunch breaks or late at night, the tuples are revisited.
- *Piggy-backing*: On pages brought into the buffer, check if any tuples need to be revisited, and then do so.

- *Explicitly scheduled revisiting*: Revisit tuples at times of expected low system load, e.g., at 2 a.m. every night.
- *Lazy*: Revisit only tuples with the temporary timestamps when a query refers to the timestamps and the permanent values are needed to process the query correctly.
- *Never*: If a query needs the permanent timestamp of a tuple, extract it from the `CommitTime` table.

The eager approach was implicitly assumed in Section 5.1. It can be implemented by using after-triggers. The approach is attractive if timestamps are often referenced in queries and modifications. However, the approach is less cost-efficient if timestamps are rarely referenced.

The "low-system-usage" approach is used in Postgres [23]; it is appropriate for an integrated architecture. However, the approach is not well-suited in a stratum because it requires scheduling of an asynchronous process based on the system load. It is hard to get this fine-level degree of control of the underlying DBMS from the stratum.

The "piggy-backing" approach is also not possible in a stratum, as the movement of pages in and out of the buffer is transparent to and cannot be controlled by the stratum.

Explicit scheduling of the revisit is a good choice if there are only current-state queries. The approach is not sufficient if there is a mixture of current-state and past-state queries within a transaction. Such queries may not execute correctly if a revisit has not yet occurred, because queries assume that the permanent timestamp values are already in place for committed tuples. The approach will have to be used in combination with the lazy approach, or the never approach, both of which are now described.

The lazy approach takes advantage of the fact that queries requiring permanent transaction timestamps can be identified by the stratum, which will then first update the transaction timestamps. This may be very cost-efficient if few queries depend on the permanent transaction timestamps for their correctness.

The never approach does not apply the timestamps from the `CommitTime` table to the temporal tables at all, but rather is applicable only if the timestamps are retained in a separate table. In the never approach, the `CommitTime` table is joined with the temporal table when referring to the transaction-time attributes. This will be expensive for large temporal tables, and is mostly useful if the transaction-time attributes are rarely referenced, say no more than two or three times in the lifetime of a tuple [23].

For the user-specified and lazy approaches to revisiting tuples, the revisiting can be done with different granularities. We see the following granularities.

- On a per-tuple basis.
- Up to a certain time.
- On a per-table basis.
- On a per-database basis.

When revisiting on a per-tuple basis, we look at each tuple the query fetches to determine if it needs to be timestamped, and do so if needed. The drawback of this approach is that it is not general. For example, it is not always possible to identify which tuples qualify for a query without first timestamping them. This happens if a query compares a timestamp to a time constant, as in “find all employees inserted after October 1, 1995:”

```
SELECT * FROM Emp WHERE T-Start >= '1995-10-01'.
```

With the up-to-a-certain-time approach we look at the query. If it implies comparisons of the transaction time of tuples to time constants, we can find the largest time constant in the query and revisit tuples that were inserted up to that point in time in all tables used by the query. This approach is also not general. For example, a query may reference transaction time without containing a comparison with a time constant. The following query compares the transaction-time attribute of different tuples:

```
SELECT * FROM Emp E1, E2 WHERE E1.T-Stop > E2.T-Stop.
```

With the per-table approach, we bring the tables referred to by the query up-to-date with respect to transaction timestamping before the query is executed. This is a general approach. However, it has the drawback of yielding non-uniform response times if the tables used in some queries have been updated frequently, but have not been revisited for a long time.

The per-database approach is similar to the per-table approach, except that it brings all tables up-to-date when a query references transaction time. This is also a general approach, but with a more non-uniform response time than the per-table approach. A query accessing but one table would be deferred until all tables are updated; the same query evaluated shortly thereafter would be much faster. This seemingly randomness in execution times is an undesirable system property.

6 Adding Valid Timestamping

In this section we discuss how the valid-time dimension is timestamped. The valid-time dimension is different from the transaction-time dimension in that the valid-time periods associated with tuples may be user-supplied. Further, for the valid-time dimension an additional special valid-time value is defined: *nobind now* [5].

6.1 An Example

We describe the general idea of adding valid time by redefining the `Emp` table from Section 5.1 to be a bitemporal table. The example will raise two questions on how to timestamp the valid-time dimension, which we then address.

To change `Emp` to a bitemporal table we issue the temporal statement [22] `ALTER TABLE Emp ADD VALIDTIME PERIOD(DATE)` on the 1st of February. The tuples already in the table are timestamped with the valid-time period [1998-02-01 - *nobind now*), where *nobind now* has a semantics similar to *until changed* for the transaction-time dimension and means "until we learn more."

We also consider a new type of modification, which is a variation on that already provided by SQL-92; in this augmented statement, the user specifies the valid-time extent of the modification.

```
VALIDTIME PERIOD '[1998-02-01 - 1998-03-01)'
UPDATE Emp
SET Dept = 'Toy'
WHERE Name = 'Joe'
```

This update refelects that Joe was in the Toy department during the month of February, 1998. We allow the temporal extent to be specified for all three types of modification statements.

To add more tuples to the bitemporal `Emp` table, we execute the temporal statements in Figure 5. On the 1st of February, we insert Kim in the Sports department. On the 2nd of February, we insert that Jill will be in the Sports department in the period [1998-02-05 - 1998-02-14). In the SQL3 proposal, this is indicated by prefixing a query with `VALIDTIME <period specification>`. On the 13th of February, we update Kim to be with the Toy department, and on the 16th of February we delete Kim. Finally, on the 27th of February, we record that John has and always will be in the Toy department.

Table 7 shows the `Emp` table resulting from the execution of these transactions. As can be seen, we include four special attributes in bitemporal tables: `V-Begin` and `V-End` for valid time, and `T-Start` and `T-Stop` for transaction time.

We next turn to how the stratum converts modifications in a temporal query language on bitemporal tables to SQL-92 modifications on SQL-92 tables. The conversion is shown in Table 8.

When we insert a tuple without a user-specified valid-time period, it is timestamped with the period [*now* - *nobind now*) in the valid-time dimension. This states that the tuple is valid from the current time until we learn more. In the transaction-time dimension, it is timestamped with the period [*temporary value* - *until changed*). For an insertion with a user-specified valid-time period, the user-specified `V-Begin` and `V-End` attributes are simply inserted into the database. The transaction-time dimension is timestamped as before.

```
-- on 1998-02-01:
INSERT INTO Emp VALUES ('Kim', 'Sports'); COMMIT;
-- on 1998-02-02:
VALIDTIME PERIOD [1998-02-05 - 1998-02-14)
INSERT INTO Emp VALUES ('Jill', 'Sports'); COMMIT;
-- on 1998-02-13:
UPDATE Emp SET Dept = 'Toy' WHERE Name = 'Kim'; COMMIT;
-- on 1998-02-16:
DELETE FROM Emp WHERE NAME = 'Kim'; COMMIT;
-- on 1998-02-27:
VALIDTIME PERIOD [0001-01-01 - 9999-12-31)
INSERT INTO Emp VALUES('John', 'Toy'); COMMIT;
```

Figure 5: Modifying the Bitemporal Table, `Emp`

`Name`	`Dept`	`V-Begin`	`V-End`	`T-Start`	`T-Stop`
Joe	Shoe	1998-02-01	*nobind now*	1998-01-06	1998-01-16
Joe	Sports	1998-02-01	*nobind now*	1998-01-16	1998-01-27
Joe	Outdoor	1998-02-01	*nobind now*	1998-01-27	*until changed*
Kim	Sports	1998-02-01	*nobind now*	1998-02-01	1998-02-13
Jill	Sports	1998-02-05	1998-02-14	1998-02-02	*until changed*
Kim	Sports	1998-02-01	1998-02-13	1998-02-13	*until changed*
Kim	Toy	1998-02-13	*nobind now*	1998-02-13	1998-02-16
Kim	Toy	1998-02-13	1998-02-16	1998-02-16	*until changed*
John	Toy	0001-01-01	9999-12-31	1998-02-27	*until changed*

Table 7: The Bitemporal Table, `Emp`

Deletions of tuples are mapped to logical deletions of currently valid tuples. A tuple is logically deleted by updating the `T-Stop` attribute to *temporary value*. The update is followed by an insertion that records the new belief that the tuple was valid in the modeled reality from the old `V-Begin` to the current-time (*now*). All explicit attribute values are copied.

For a delete statement with a user-specified valid-time period, two insertions are needed, one for the portion of the original tuple's valid-time that is *before* the user-specified period, and one for the portion after the user-specified period. The last statement, the update, terminates the original tuple.

An update, e.g., that updates Kim to be in the Toy department on the 13^{th} of February, is a temporal delete of the old values followed by a temporal insert of the new values. Similarly, an update with a user-specified valid-time period is a delete

Temporal statement: `INSERT INTO Emp VALUES (`*new name*`, `*new dept*`)`
  Corresponding SQL-92 statement:
    `INSERT INTO Emp VALUES (`*new name*`, `*new dept*`, `*now*`, `*nobind now*`,`
      temporary value`, `*until changed*`)`

Temporal statement:
  `VALIDTIME PERIOD [`*Start*` - `*Stop*`)`
  `INSERT INTO Emp VALUES (`*new name*`, `*new dept*`)`
  Corresponding SQL-92 statement:
    `INSERT INTO Emp VALUES (`*new name*`, `*new dept*`, `*Start*`, `*Stop*`,`
      start value`, `*until changed*`)`

Temporal statement: `DELETE FROM Emp WHERE` *Predicate*
  Corresponding SQL-92 statements:
    `INSERT INTO Emp`
    `SELECT Name, Dept, V-Begin, `*now*`, `*temporary value*`, `*until changed*
    `FROM Emp WHERE `*Predicate*` AND T-Stop = `*until changed*
    `AND V-Begin < `*now*` AND `*now*` < V-End;`
    `UPDATE Emp SET T-Stop = `*temporary value*
    `WHERE `*Predicate*` AND T-Stop = `*until changed*
    `AND V-Begin < `*now*` AND `*now*` < V-End`

Temporal statement:
  `VALIDTIME PERIOD [`*Start*` - `*Stop*`) DELETE FROM Emp WHERE `*Predicate*
  Corresponding SQL-92 statements:
    `INSERT INTO Emp`
    `SELECT Name, Dept, V-Begin, `*Start*`, `*start value*`, `*until changed*
    `FROM Emp WHERE `*Predicate*` AND T-Stop = `*until changed*
    `AND V-Begin < `*Start*` AND `*Start*` < V-End;`
    `INSERT INTO Emp`
    `SELECT `A_1`, ..., `A_n`, `*Stop*`, V-End, `*start value*`, `*until changed*
    `FROM Emp WHERE `*Predicate*` AND T-Stop = `*until changed*
    `AND V-Begin < `*Stop*` AND `*Stop*` < V-End;`
    `UPDATE Emp SET T-Stop = `*stop value*
    `WHERE `*Predicate*` AND T-Stop = `*until changed*
    `AND V-Begin < `*Stop*` AND `*Start*` < V-End`

Temporal statement: `COMMIT`
  Corresponding SQL-92 statements:
    $c \leftarrow$ `CURRENT_DATE;`
    `INSERT INTO Time VALUES (`*transaction-id*`, `c`);`
    `UPDATE Emp SET T-Start = `c` WHERE `*tuple inserted by transaction-id*`;`
    `UPDATE Emp SET T-Stop = `c` WHERE `*tuple inserted by transaction-id*`;`
    `DELETE FROM Time WHERE TID = `*transaction-id*`;`
    `COMMIT`

Table 8: Mapping Statements on Bitemporal Tables into Equivalent SQL-92 Statements

with a user-specified valid-time period followed by an insert with a user-specified valid-time period.

When the user enters commit (the fifth row of Table 8), the appropriate `T-Start` and `T-Stop` attributes are updated, similarly to what occurred in Table 6.

If the statements described in Table 8 are applied to the transactions in Figure 5, Table 7 results.

We now address in turn three questions with respect to timestamping the valid-time dimension: How is *now* represented, what should `CURRENT_DATE` be mapped to, and how is *nobind now* represented?

6.2 Handling *now*

Section 3 illustrated that a single value for *now* must be used throughout a transaction in order to avoid violating the consistency property. The question is, then, which value to use for *now*. We see the following possibilities.

- The start time of the transaction.
- The time of the first modification in the transaction.
- The commit time of the transaction.

It has been shown that using the start time of the transaction can lead to the current time appearing to move backwards [10]. It has also been shown that using the time of the first update can cause a violation of the isolation property, even when two-phase locking is used [10]. Both problems occur because neither the start times nor the times of the first updates in transactions reflect the commit order of the transactions.

It is also the case that using the start time of the transaction can result in the violation of the ACID properties, because the transactions are no longer serializable. Consider Figure 2 again. Transaction T_1 starts on the 4^{th} of January and commits on the 12^{th} of January. On the 10^{th} of January, T_1 updates Jim to be in the Outdoor department. Transaction T_2 starts after T_1, on the 7^{th} of January, and commits before T_1, on the 9^{th} of January. On the 8^{th} of February, T_2 updates Jim to be in the Sports department. Assume Jim was inserted in the Shoe department on the 2^{nd} of January and we are using the start time of the transaction for *now*. The tuples relating to Jim after the execution of T_1 and T_2 are then as shown in Table 9.

First, notice that the fourth tuple is timestamped with the valid-time period `[`1998-01-07 `-` 1998-01-04`)`; the value of `V-Begin` is larger than the value of `V-End`. Second, notice that the second and the fifth tuples are both in the current transaction-time state (the current time is 1998-01-04) and have overlapping valid-time periods, `[`1998-01-04 `-` 1998-01-07`)` and `[`1998-01-07 `-` *nobind now*`)`, respectively, even though Figure 2 shows that Jim was only in a single department at any point in time.

Name	Dept	V-Begin	V-End	T-Start	T-Stop
Jim	Toy	1998-01-04	1998-01-12	1998-01-02	*until changed*
Jim	Toy	1998-01-04	1998-01-07	1998-01-02	1998-01-07
Jim	Sports	1998-01-07	*nobind now*	1998-01-07	1998-01-04
Jim	Sports	1998-01-07	1998-01-04	1998-01-04	*until changed*
Jim	Outdoor	1998-01-04	*nobind now*	1998-01-04	*until changed*

Table 9: Tuples for Jim in the Bitemporal Table, `Emp`

The result of the two transactions T_1 and T_2 does not correspond to any serial execution of the transactions. In a serial execution, the two problems discussed above cannot occur because the start times will reflect the commit order of the transactions.

Because using the start time as well as using the time of the first modification for *now* can lead to violation of the isolation property of transactions, we elect to use the commit time for *now*. For transactions to be able to see their own modifications to the valid-time dimension, modified tuples are given a *temporary value* and are revisited, to update the *temporary value*, after user commit. We use the same temporary value for *now* as for *until changed*.

Using the commit time eliminates both the problem of current time moving backwards and the violation of the isolation property. The drawback is that modified tuples must be revisited, which requires extra resources. However, in the case of bitemporal tables, the tuples must be revisited anyhow, to apply the permanent transaction-time timestamps. Thus, the number of tuples that are to be revisited is not increased.

However, for valid time revisitation raises a new problem. Consider the example in Figure 6. This transaction inserts James in the Shoe department and then

```
-- on 1998-02-20:
INSERT INTO Emp VALUES ('James', 'Shoe');
VALIDTIME PERIOD [1998-02-01 - 1998-02-21)
    DELETE FROM Emp WHERE Name = 'James';
COMMIT;
```

Figure 6: A Race Condition in a Delete Transaction

deletes James from the the Sports department in the period `[`1998-02-01 `-` 1998-02-21`)`. This leads to a race condition. If the transaction commits on the 20^{th} of February, the delete will have an effect on the inserted tuple. If the clock ticks and the transaction actually commits on the 21^{st} of February, the update has no effect on the inserted tuple because the valid-time period associated with the insertion no longer overlaps with the valid-time period specified in the delete. When the delete is actually executed, the commit time is not known, and we cannot determine what to do.

The general problem is illustrated in Figure 7. Here, the circles (filled or non-filled) represent timestamp values given explicitly in a modification statement, and '$\times$' represents the temporary value of *now* used in modifications. The sequence of modifications within a transaction that can cause the problem is an insertion (or an update) of one or more tuples using *now* (indicated by the two "a"'s in Figure 7), followed by a deletion (or an update) of the same tuples, using an explicitly given period that overlaps with the temporary value of *now*(indicated by the two "b"'s in Figure 7).

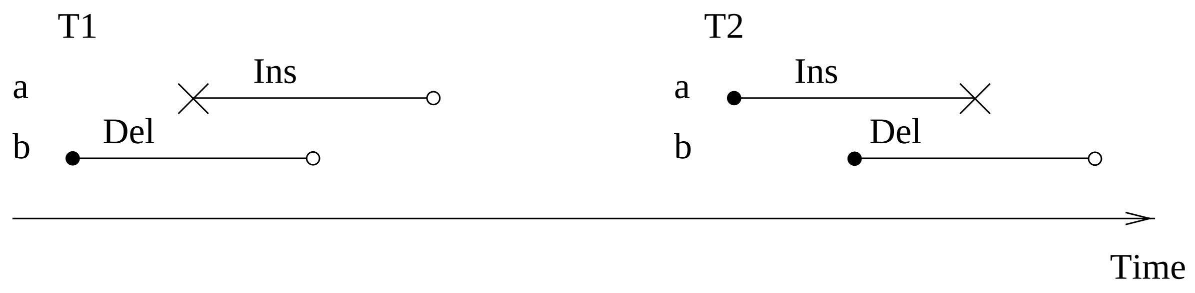


Figure 7: The General Problem Using the Commit Time for *now*

Syntactic analysis may be applied to detect when the problem may occur. We have to store the smallest explicit timestamp value for any period [*now* - *explicit timestamp*) within a transaction. If the smallest explicit timestamp value is smaller than the commit time, the problem is present. We have not found any way of solving the problem and getting a clear semantics. Therefore, when this situation occurs, the transaction is declared illegal and is rolled back.

6.3 Handling `CURRENT_DATE`

The presence of `CURRENT_DATE` (and `CURRENT_TIME` and `CURRENT_TIME-STAMP`) in queries and in modification statements causes some problems. Users expect that this value is identical to teh value of *now* that is stored in tuples, yet that latter value is the commit time for the transaction. So, we must map usages of `CURRENT_DATE` into expressions consistent with the value of *now*.

We will address the handling of `CURRENT_DATE` in two steps. First, we handle this function in the context of modifications, discussing implications of several alternative approaches. Second, we consider the function in queries and again discuss implications of various approaches.

`CURRENT_DATE` in Modifications

A user may specify arbitrary valid-time periods in modification statements. The combination of using the commit time for both *now* and `CURRENT_DATE` and the constraint that `V-Begin` must be smaller than `V-End` can cause problems in transactions, as shown in Figure 8. Here we use a period constructor, which takes two SQL-92 datetime expressions as arguments.

```
-- on 1998-02-20:
VALIDTIME PERIOD [CURRENT_DATE, DATE '1998-02-21')
    INSERT INTO Emp Values ('James', 'Shoe');
COMMIT;
```

Figure 8: A Race Condition in an Insert Transaction

The transaction inserts a tuple and timestamps it with the period [*now* - 1998-02-21). This leads to a race condition. If the transaction commits on the 20th of February, no anomalies occur. However, if the transaction does not commit before the 22th of February, the `V-Begin` will be larger than the `V-End` value, which is not allowed.

A possible approach to addressing this problem is for the revisit step to remove tuples with erroneous valid-time periods. However, this does not work because subsequent statements in the transaction might reference these tuples in the meantime, corrupting the result of the transaction. Instead, we adopt the solution of the previous section, of identifying the smallest explicit timestamp value for any period [*now* - *explicit timestamp*) within the transaction (here, the explicit timestamp is 1998-02-21). If that value is smaller than the commit time, the transaction is considered illegal and is rolled back.

We next examine the consequences of using a single value for `CURRENT_DATE` and show it impacts the query rewriting performed in the stratum to retain the semantics of SQL-92 queries.

In Figure 9, transaction T_1 inserts Joe in the Shoe Department on the 8th of February. On the 19th of February, T_1 executes the modification M that deletes all employees in the Shoe Department: `DELETE FROM Emp WHERE Dept = 'Shoe'`. Transaction T_2 inserts Tim in the Shoe Department and transaction T_3 inserts Lee in the Shoe department for the period [1998-02-10 - 1998-02-25). The question is, what effect does M have on the insertions made by T_2 and T_3?

The content of the `Emp` table on the 19th of February, when M is ready to execute, is shown in Table 10. Remember, T_1 has not committed yet, which means `V-Begin` and `T-Start` of the first tuple have a temporary value, 1998-02-08, because this is when the value is first needed in T_1. To emphasize this, these temporary values are shown in italics.

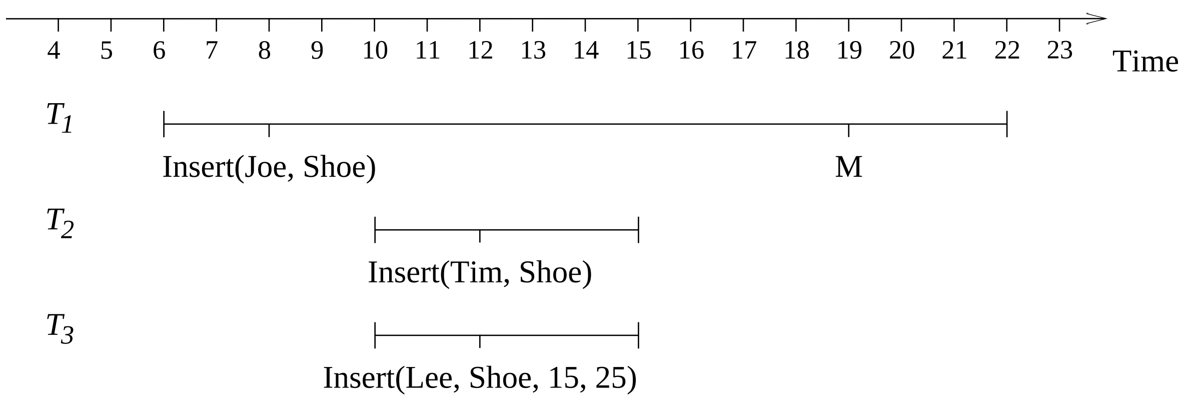


Figure 9: Using a Single Value for `CURRENT_DATE` in Queries

Name	Dept.	`V-Begin`	`V-End`	`T-Start`	`T-Stop`
Joe	Shoe	*1998-02-08*	*nobind now*	*1998-02-08*	*until changed*
Tim	Shoe	1998-02-15	*nobind now*	1998-02-15	*until changed*
Lee	Shoe	1998-02-15	1998-02-25	1998-02-15	*until changed*

Table 10: The Bitemporal Table `Emp` at 1998-02-19

T_1 now executes the modification M. Referring to the delete in Table 8 and assuming *nobind now* has a reasonable value, the first tuple is deleted because `V-Begin` is smaller than or equal to the value of *now*. The second tuple has a `V-Begin` larger than the value of *now*. However, it must be deleted to fulfill the SQL-92-query requirement. The tuple is inserted by a transaction that commits before the deletion is applied. The third tuple is also inserted by a transaction that commits before the deletion is applied, and the tuple is in the current bitemporal state when T_1 commits. This could indicate that the third tuple should also be deleted. However, the valid-time period associated with the tuple does not overlap with the value of *now*.

The impact of using one value for *now* in transactions can be summarized as follows. Tuples that have a `V-End` value of *nobind now* will be affected by modifications in a transaction, even if the value of *now* used in the transaction is smaller than the `V-Begin` attribute values of the tuples. This ensures that SQL-92 like modifications on temporal tables have the expected results. For tuples with an explicit value in `V-End`, read-level consistency within a transaction is provided.

It is important that the valid-time periods associated with tuples are checked for whether the `V-End` is equal to *nobind now* or the valid-time periods overlap with the value of *now* used in queries. Doing just the latter, which may seem sufficient, can violate the requirement of temporal upward compatibility, stating that SQL-92 queries should have the same effect on snapshot as on bitemporal tables.

CURRENT_DATE in Queries

Timestamping after commit with revisitation presents the same problem for the valid-time dimension as for the transaction-time dimension when the user modifies the database and then queries it with explicit reference to valid time. The handling of CURRENT_DATE in queries is further complicated by the user being allowed to insert tuples with valid-time attributes in the future.

There are two approaches for using CURRENT_DATE in queries. (a) We can use the same value in the entire transaction, e.g., the time when a value of CURRENT_DATE is first needed. (b) We can permit the use of different values for CURRENT_DATE in different queries in the same transaction, which may be obtained simply by leaving invocations of CURRENT_DATE as are in queries.

When using a single value for CURRENT_DATE in queries, we cannot use the same value as for modification, because this time is the commit time, which is unknown until the transaction actually commits. Instead assume we use the time when CURRENT_DATE is first needed, which can be determined syntactically, and consider the transaction in Figure 10.

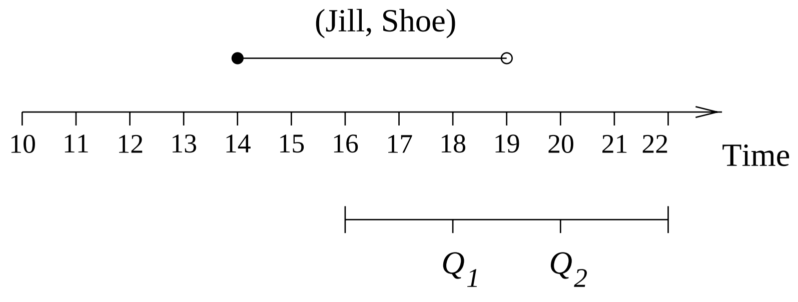


Figure 10: CURRENT_DATE in Queries

This transaction starts on the 16th of January and commits on the 22nd of January. The database includes a tuple recording Jim is in the Shoe department in the period [1998-01-14 - 1998-01-19). Now assume that query Q_1, executed at the 18th of January, asks for all the employees currently in the Toy department. Q_1 must use a value for CURRENT_DATE and this is the time, it is first needed, so CURRENT_DATE will be instantiated to the 18th of January. On the 20th of January, query Q_2 asks for all the employees currently in the Shoe department. Because we use only one value for CURRENT_DATE in a transaction, we return the 16th of January, and Q_2 will return that Jim is in the Shoe department, even though the query is executed at the 20th of January, and we have recorded that Jim is in the Shoe department only during the period [1998-01-14 - 1998-01-19).

Next, we consider the alternative of using multiple values for CURRENT_DATE in queries within a transaction. For each statement within a query, we retain CURRENT_DATE as is, with each invocation possibly yielding a different value.

Consider Figure 10 again. This approach evaluates `CURRENT_DATE` to the 18th of January and the 20th of January in queries Q_1 and Q_2, respectively. This also causes problems. For example, if Q_1 and Q_2 are the same query retrieving all employees currently in the Shoe department. On the 18th of January, the query will return that Jill is in the Shoe department, and at the 20th of January, the query will return there are no employees in the Shoe department. Two identical queries with no intermediate modifications within the same transaction should not return different results. The disappearance of the tuple recording Jill in the Shoe department is similar to the non-repeatable read problem [12, pp. 380] and constitutes a violation of the isolation property of transactions. Non-repeatable reads is a multi-user problem in conventional database systems; here, a non-repeatable read can also appear in a single-user system.

The problems using a single value or multiple values for `CURRENT_DATE` in a transaction are caused by the possibility of inserting tuples with a valid time into the future; this is not allowed for transaction-time. We have to choose between disallow insertion of tuples with valid-times into the future or use one of the approaches mention above.

Disallowing valid-times into the future is a serious restriction on temporal databases: it will make them useless for many applications such as, e.g., planing applications. This option is therefore ruled out. We must then choose to use one value or multiple values for `CURRENT_DATE` in queries. We chose the former alternative, because using multiple values can cause non-repeatable reads to occur in SQL-92 queries on temporal tables.

6.4 Handling *nobind now*

As discussed in Section 6.1, there is also the special value *nobind now* in the valid-time dimension. Because this value is not part of the SQL-92 timestamp domain, an SQL-92 value for representing it must be identified.

We can use `NULL` or a value from the time domain. The problem with using `NULL` for *nobind now* is that it is then not possible to store “real” `NULL` values in the valid-time dimension. Problems with picking a value from the time domain are that this restricts the time domain and that we must handle the representative value specially.

7 A Specific Proposal

This section motivates and presents an overall approach to timestamping the valid-time and transaction-time dimensions. The proposal is based on two assumptions. We are focusing on the stratum approach to implement a temporal DBMS; and to be specific, we are using Oracle 8 as the underlying DBMS in the stratum approach.

For timestamping the transaction-time dimension, we use timestamping after commit with revisitation. For the *temporary value* of the commit time, we use one value throughout a transaction. We choose to use the time of the first modification statement or the time of the first statement that references `CURRENT_DATE`, whichever comes first. This is the constant value closest to the commit time we can use within a transaction. For the value of *until changed*, we choose the largest value in the time domain. We could also use `NULL`. However, this may invalidate the use of indexes in Oracle [6].

For the valid-time dimension, we use the commit time as the value of *now* in modifications. When using the commit time, we again need a temporary value for *now* within the transaction. The temporary value for *now* is the same as the *temporary value* used for transaction time. We also use this value for `CURRENT_DATE` in queries and modification statements.

Again, because Oracle handles `NULL` badly in connection with indexes, we do not use `NULL` as the value of *nobind now*. Instead, we use the second largest value in the time domain, as the largest value in the time domain is commonly used to represent *forever*. The representations of the special temporal values can be seen in Table 11.

Special Value	Representation
nobind now	9999-12-30
beginning	0001-01-01
forever	9999-12-31
until changed	9999-12-31

Table 11: Representation of the Special Temporal Values

When the timestamp attributes occur in a `SELECT` or a `WHERE` clause, a `CASE` statement is introduced to ensure that the special temporal values *nobind now* and *until changed* are interpreted correctly and to ensure that their representations remain hidden from the user. As an example, `WHERE V-End < '1998-01-10'` is converted to `WHERE CASE WHEN V-End = '9999-12-30' THEN CURRENT_TIME ELSE V-End END < '1998-01-10'`. This approach is consistent with the recommendations by Clifford et al. [5].

In the following three tables, we provide the specifics for mapping temporal statements. We use the `Emp` table as an example.

Table 12 shows the mapping of the `CREATE TABLE` and `INSERT` statements. The first row shows the mapping of a `CREATE TABLE` statement. For a bitemporal table, four attributes are added (as an aside, more attributes are added in Section 8). The second row shows that at the time of the first modification or the first use of `CURRENT_DATE`, we fix the value of *temporary value* within a transaction.

The value of *temporary value* is used in the third row, which gives the mapping of an `INSERT` statement and uses several of the special temporal values. The fourth row shows the mapping of an `INSERT` statement with a user-specified period. The `IF` statement is used to find the smallest explicit timestamp in periods of the form `[`*now* `-` *explicit timestamp*`)` within a transaction. The value *smallest_explicit_timestamp* is used to detect race conditions, as discussed in Section 6.3.

Temporal statement:

```
CREATE TABLE Emp (Name VARCHAR(20),  Dept VARCHAR(20))
AS VALIDTIME PERIOD(DATE) AND TRANSACTIONTIME
```

Resulting statement:

```
CREATE TABLE Emp (  Name VARCHAR(20), Dept VARCHAR(20),
V-Begin DATE, V-End DATE,  T-Start DATE, T-Stop DATE)
```

Temporal statement: first modification or use of `CURRENT_DATE`

Resulting statement: *temporary value* ← `CURRENT_DATE`

Temporal statement: `INSERT INTO Emp VALUES (`*new name*`,` *new dept*`)`

Resulting statement:

`INSERT INTO Emp VALUES (`*new name*`,` *new dept*`,` *temporary value*`,` *nobind now*`,` *temporary value*`,` *until changed*`)`

Temporal statement:

`VALIDTIME PERIOD [`*Start* `-` *Stop*`)`
`INSERT INTO Emp VALUES (`*new name*`,` *new dept*`)`

Resulting statement:

`IF (`*Start* is *now* `AND` *Stop* is an *explicit timestamp* `AND` *smallest_explicit_timestamp* $>$ *Stop*`)` *smallest_explicit_timestamp* ← *Stop*
`INSERT INTO Emp VALUES (`*new name*`,` *new dept*`,` *Start*`,` *Stop*`,` *temporary value*`,` *until changed*`)`

Table 12: Mapping Create Table and Insert Statements on Bitemporal Tables

Table 13 covers `DELETE` statements. In the first row, a `DELETE` statement without a user-specified period is shown. The statement is mapped to an `INSERT` of a new tuple followed by an `UPDATE` of the existing tuple. The last line in the `WHERE` clauses for the `INSERT` and `UPDATE` statements is used to identify the tuples inserted by other transactions that must be logically deleted to fulfill the temporal upwards compatibility requirement, as discussed in Section 6.3. The second row in the table gives the mapping of a `DELETE` statement with a user-specified period. As for insertions with user-specified periods, the `IF` statement keeps track (within a transaction) of the smallest explicit timestamp used in periods of the form `[`*now* `-` *explicit timestamp*`)`. Note that the `DELETE` with a user-specified period may result in two new tuples being added to the table and a single tuple being updated. This happens if, for example, the period `[`20 `-` 30`)` is deleted from a tuple timestamped with the valid period `[`10 `-` 40`)`.

In the implementation used in the performance evaluation presented in Section 8, both `DELETE` statements in Table 13 are accomplished using cursors. This is done for efficiency reasons: when using cursors, all tuples to delete can be retrieved by evaluating a single `WHERE` clause, instead of the two and three `WHERE` clauses used in Table 13.

```
Temporal statement: DELETE FROM Emp WHERE Predicate
  Resulting statements:
    INSERT INTO Emp SELECT Name, Dept, V-Begin, temporary value,
                                              temporary value, until changed
    FROM Emp WHERE Predicate AND T-Stop = until changed
    AND ((V-Begin <= temporary value AND temporary value < V-End)
    OR (V-Begin <= T-Start AND V-End = nobind now));
    UPDATE Emp SET T-Stop = temporary value
    WHERE Predicate AND T-Stop = until changed
    AND ((V-Begin <= temporary value AND temporary value < V-End)
    OR (V-Begin = T-Start AND V-End = nobind now));
Temporal statement:
  VALIDTIME PERIOD [Start - Stop) DELETE FROM Emp WHERE Predicate
  Resulting statements:
    IF (Start is now AND Stop = is an explicit timestamp AND
        smallest_explicit_timestamp > Stop) smallest_explicit_timestamp ← Stop
    INSERT INTO Emp SELECT Name, Dept, V-Begin, Start,
                                              temporary value, until changed
    FROM Emp WHERE Predicate AND T-Stop = until changed
    AND V-Begin < Start AND Start < V-End;
    INSERT INTO Emp
    SELECT Name, Dept, Stop, V-End, temporary value, until changed
    FROM Emp WHERE Predicate AND T-Stop = until changed
    AND V-Begin < Stop AND Stop < V-End;
    UPDATE Emp SET T-Stop = temporary value WHERE Predicate
    AND T-Stop = until changed AND V-Begin < Stop AND Start < V-End;
```

Table 13: Mapping Delete Statement on Bitemporal Tables

Finally, Table 14 shows the mapping of transaction start and eager and lazy commit. When a transaction starts, we initialize the variable *smallest_explicit_timestamp* to the maximum value in the time domain (the variable is local to each transaction). For both eager and lazy timestamping, the variable is used to determine if any race conditions occurred, requiring that the transaction be rolled back, as indicated by the `IF` statements in Table 14.

When a transaction commits and we are using eager timestamping and no race conditions occur, we find the value of *now*, and all tuples modified by the transaction are revisited. For lazy timestamping, this is a two-stage process. First, when the transaction commits, the commit time is stored in the table `CommitTime`. Second, the revisit step is scheduled in a separate transaction, shown in the fourth row of Table 14. When the revisit step is executed, all tuples modified since the last revisit are updated with the permanent timestamps for *now* by using the commit times stored the `CommitTime` table. When the timestamps have been applied, the `CommitTime` table is cleaned up.

As for the `DELETE` statement, the `COMMIT` statements used in the performance study are implemented using cursors.

```
Temporal statement: start transaction
  Resulting statement: smallest_explicit_timestamp ← 9999-12-31
Temporal statement: COMMIT (eager)
  Resulting statement(s):
    now ← CURRENT_DATE;
    IF (smallest_explicit_timestamp < now) ROLLBACK;
    ELSE
    UPDATE Emp SET V-Begin = now WHERE tuple modified by this transaction;
    UPDATE Emp SET V-End = now WHERE tuple modified by this transaction;
    UPDATE Emp SET T-Start = now WHERE tuple modified by this transaction;
    UPDATE Emp SET T-Stop = now WHERE tuple modified by this transaction;
    COMMIT;
Temporal statement: COMMIT (lazy)
  Resulting statements:
    now ← CURRENT_DATE;
    IF smallest_explicit_timestamp < now ROLLBACK
    ELSE INSERT INTO CommitTime VALUES (transaction-id, now);
Temporal statement: timestamp_table
  Resulting statements:
    UPDATE Emp SET V-Begin =
    SELECT Commit-Time FROM CommitTime
    WHERE TID = TID modified V-Begin WHERE V-Begin needs revisiting;
    UPDATE Emp SET V-End =
    SELECT Commit-Time FROM CommitTime
    WHERE TID = TID modified V-End WHERE V-End needs revisiting;
    UPDATE Emp SET T-Start =
    SELECT Commit-Time FROM CommitTime
    WHERE TID = TID modified T-Start WHERE T-Start needs revisiting;
    UPDATE Emp SET T-Stop =
    SELECT Commit-Time FROM CommitTime
    WHERE TID = TID modified T-Stop WHERE T-Stop needs revisiting
    DELETE FROM CommitTime;
```

Table 14: Mapping Start and Commit of Transactions on Bitemporal Tables

The WHERE clauses for the mapping of the COMMIT statements in Table 14 are deliberately vague. This is because two open issues remain: (1) how to associate transaction-id's with tuples (discussed in Section 5.3), and (2) which of the revisiting approaches identified in Section 5.5 is the most cost-efficient and thus should be adopted. We now conduct a performance study resolving these issues.

8 Performance Evaluation

Section 5.3 discussed how transaction-ids can be associated with tuples, and in Section 5.5 we presented a spectrum of approaches for scheduling the revisiting step. Some of these approaches are viable only within the DBMS; others apply equally well to situations with applications directly handling time-varying data or with temporal support being implemented in a stratum. After having stated the objectives of the performance study and described the experimental setup in Sections 8.1 and 8.2, Section 8.3 proceeds to evaluate the performance of the various approaches to associating transaction-ids with tuples. We then evaluate the two revisiting approaches anchoring the spectrum, the eager and lazy approaches; both are well-suited for implementation in a stratum. This is done in Sections 8.4 and 8.5.

8.1 Objectives of the Performance Evaluation

We first attempt to determine how transaction-ids should be associated with tuples to make revisiting efficient. The transaction-ids identify which tuples must be revisited to apply correct, permanent timestamps. It is therefore essential for the performance of the revisiting step that given a transaction-id, it is easy to identify exactly which tuples to timestamp. Because the issue of associating transaction-ids with tuples is orthogonal to the choice of revisiting approach, we simply use the eager approach.

With the performance study of revisiting approaches, also presented in this section, we want to answer the following two questions.

1. For different transaction sizes, which revisiting approach is most cost efficient, the eager or the lazy approach?
2. How expensive is the revisiting step compared to the actual execution of the transaction?

The answer to the first question is important because it affects temporal DBMS implementation and transaction design. It is more complicated to implement and schedule the revisiting of modified tuples in the lazy approach, compared to the straightforward revisiting of tuples in the eager approach. If lazy revisitation does not perform better than eager revisitation, e.g., by allowing us to postpone the revisiting step to be done during off hours, there is no reason to add the extra complexity of the lazy approach to the temporal DBMS implementation.

Further, the answer to the first question is important for the transaction designer to be able to tune applications. If one revisitation approach is superior for certain transaction sizes and the other approach is superior for other transaction sizes then for a given transaction size, the designer can determine which approach is the most cost-efficient in a particular situation.

The second question is important for transaction performance reasons. Using timestamping after commit adds the revisiting step to the cost of executing a transaction.

We investigate the questions by running a set of experiments using the stratum architecture described in Section 2. This architecture is well-suited for the evaluation because we can use an existing commercial relational DBMS, thus obtaining a realistic picture of transaction performance.

8.2 Performance Evaluation Setup

We use the Oracle 8.0.4 DBMS running on a SUN UltraSparc-2. Our test database contains a single bitemporal table `Emp` that has two explicit attributes, `NameId` and `DeptId`, of type `INTEGER`, recording which employees are affiliated with which departments. The four timestamp attributes `V_BEGIN`, `V_END`, `T_START`, and `T_STOP` capture valid and transaction time.

There are 5,000 tuples in the current state of the `Emp` table; this number is constant. We simulate the update activity of an application over a number of months. For each simulated month, we insert 5%, delete 5%, and update 10% of the current state. We run our experiments starting with an 18-month old table. This table contains approximately 822,000 tuples, which occupy approximately 42MB. Our page size is 8KB, and the buffer size of the database is 1.5 MB. The entire current state does not fit in the buffer because the table is stored in `T-Start` order and tuples in the current state can have `T-Start` values within the entire transaction-time period (18 months) of the table.

The tests are performed by executing a total of 2000 modifications as a series of transactions where we vary the transaction size (m, the number of modifications in each transaction). For the lazy approach, we also vary the *inter-visitation interval*, that is, the number (n) of transactions between revisiting tuples. The elapsed time is measured using Oracle's `DBMS_UTILITY.GET_TIME` function [9] before the first transaction starts and then after each user commit and system commit for the eager approach. For the lazy approach, we also measure the time before a revisit. As is customary for (non-simulation based) performance measurements on database systems [7, 17], we only report on the elapsed-time usage. The numbers we report here are the averages of our measurements for a single test series, i.e., execution of the 2000 modifications with fixed values for n and m. Repeated executions of the test series showed little variation.

Note that only one table is needed for the test setup because we are examining modifications. A modification statement, by its very nature, concerns only a single table.

8.3 Number of Transaction-id Attributes on Bitemporal Tables

Section 5.3 described how transaction-ids can be associated with tuples. Two overall approaches were discussed, namely storing the transaction-ids in the timestamp attributes by encoding the transaction-ids as the smallest values in the time domain, versus storing the transaction-ids in separate attributes.

We proceed to evaluate the performance of the following three approaches for associating transaction-ids with tuples.

- Storing the transaction-ids in the timestamp attributes.
- Storing all transaction-ids in a single, separate attribute.
- Storing each transaction-id in a separate attribute.

The first approach adds no extra attributes to a bitemporal table. The second and third approaches add one and four extra transaction-ids attributes to a bitemporal table, respectively. Due to the number of attributes added, we refer to the three approaches as the *zero-tid*, the *one-tid*, and the *four-tids* approaches. The schemas of the tables used in the performance study are shown below.

```
TABLE Emp_0 (NameId, DeptId, V_BEGIN, V_END, T_START, T_STOP)

TABLE Emp_1 (NameId, DeptId, V_BEGIN, V_END, T_START, T_STOP,
             TIDS)

TABLE Emp_4 (NameId, DeptId, V_BEGIN, V_END, T_START, T_STOP,
             V_BEGIN_TID, V_END_TID, T_START_TID, T_STOP_TID)
```

Although we are only concerned with modifications in this performance study, we have chosen to use an indexing scheme that is suitable when both queries and modifications are considered. For all three approaches, we used a composite B^+-tree index on the `NameId`, `V_BEGIN`, and `V_END` attributes. This index speeds up modifications and queries.

To be able to identify which tuples to timestamp, we add for the zero-tid approach two B^+-tree indexes, on the `T_START` and `T_STOP` attributes, respectively. An index on both of these attributes is needed because we must check both the `T_START` and `T_STOP` attributes for the smallest values in the time domain. Tests without indexes on both the `T_START` and `T_STOP` attributes show results orders of magnitude slower because full table scans are performed. For the one-tid approach, we add a B$^+$-tree index on the `TIDS` attribute, and for the four-tids approach, we add a B$^+$-tree index on each of the `T_START_TID` and `T_STOP_TID` attributes. Test show that also for the four-tids approach, indexes on both the `T_START_TID` and `T_STOP_TID` attributes are needed to avoid full table scans.

To perform the tests, we load the 822,638 tuples into each of the tables and execute 2000 modifications for each table in each test series. After a test series is executed, all three tables are restored to the initial state. In a test series, the transaction size (*m*) is fixed, and to avoid artificially high buffer hit rates, we interleave the transactions so that first a transaction is executed on the `Emp_0` table, then one on the `Emp_1` table, then on on the `Emp_4` table, and so on.

Figure 11 shows the average elapsed-time per transaction for various transaction sizes, for the three approaches to associating transaction-ids with tuples.

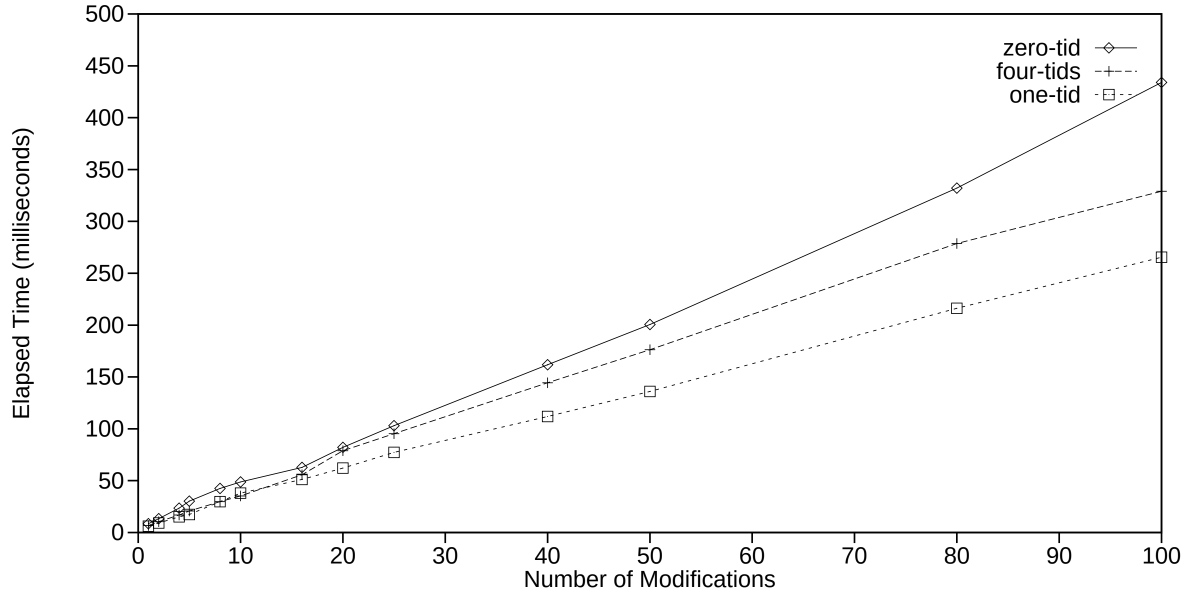


Figure 11: The Number of Transaction-ids on Tables

As shown in Figure 11, the one-tid approach is always faster than the two other approaches. For transactions of size larger than or equal to four, the one-tid approach is approximately 40% faster than the zero-tid approach and approximately 20% faster than the four-tids approach. The one-tid approach becomes percent-wise slightly better with increasing transaction size.

These results can be explained as follows. For the actual execution of the modifications, the one-tid and the four-tids approaches are equally fast because the data content, indexes used, and modification logic are identical. Storing the transaction-ids in the timestamp attributes, in the zero-tid approach, makes the cost of modifying tuples higher because the modification logic becomes more complicated: if timestamp values that occur in tuples identify transactions (and are not “real” values), special handling is necessary in the predicates involving the timestamp values. However, the major differences in performance are caused by the revisiting and timestamping costs. The one-tid approach is the fastest because tuples to timestamp can be efficiently identified by the index on the `TIDS` attribute. This index only indexes tuples that must be timestamped. The four-tids approach is slower because two different indexes must be used. Again these two indexes only index tuples that must be timestamped. Finally, storing the transaction-ids in the timestamp attributes results in the slowest timestamping because identifying the tuples to timestamp requires using two larger indexes, over all tuples in the table.

With respect to the storage usage of the tables (not including storage used for indexes) the `Emp_0` table is the smallest because no extra attributes are added. The `Emp_1` and `Emp_4` tables are 2.5% and 9.1% larger than the `Emp_0` table, respectively.

Based on this performance study, we will in the following use the one-tid approach for associating transaction-ids with tuples.

8.4 Eager versus Lazy Revisitation

The next experiment measures the cost of executing transactions using eager versus lazy revisitation strategies. To be able to compare the cost for various transaction sizes and inter-visitation intervals, we report the cost on a per-modification basis.

We use the one-tid approach discussed in the previous section with the accompanying indexing scheme described there. To verify this indexing scheme we ran a set of six test queries and six test modifications on five different indexing schemes. The chosen indexes provided the best overall response time. The first index was needed to speed up queries; the latter index sped up modifications.

Figure 12 shows the total elapsed time per modification for the eager and lazy approaches. The elapsed time is shown for varying transaction sizes (m). For the lazy approach, we also vary the inter-visitation interval (n). The time taken to revisit tuples in the lazy approaches have been divided equally among the transactions executed since the last revisit. For example, if the inter-visitation interval is five, we have added to each transaction's total elapsed-time one fifth of the elapsed time for revisiting. Note that this is possible because the transaction size is fixed within a test series.

Figure 12 shows that using lazy revisitation immediately after a transaction containing only one modification is approximately 11% more expensive than eager revisitation for the same transaction size. Lazy revisitation is more expensive because the revisiting is done using a separate transaction, whereas eager revisitation is done in the transaction that also did the modifications. However, for n larger than two, lazy timestamping is more efficient.

In general, Figure 12 shows that the combination of small transactions and revisiting often is expensive on a per modification basis. This is due to the extra

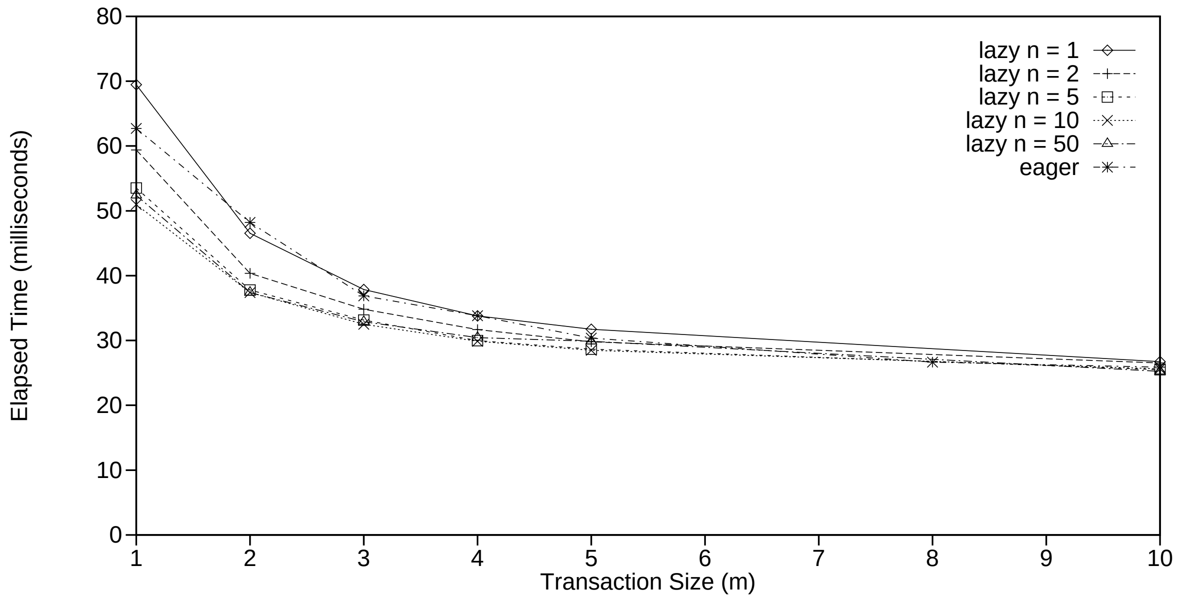


Figure 12: Elapsed time Per Modification for Eager and Lazy Revisitation

transaction executed for lazy revisitation. However, if a transaction contains more than just five modifications, then lazy revisitation is less than 10% cheaper than eager revisitation, independently of the inter-visitation interval. For larger transaction sizes and larger inter-visitation intervals, the costs of eager and lazy revisitation are almost identical.

We also ran experiments with inter-visitation intervals (n) of up to 200 transactions for transaction sizes (m) of up to 5 modifications; and we experimented with the inter-visitation intervals of up to 10 transactions for transaction sizes of up to 100 modifications. These experiments were consistent with the trend illustrated by Figure 12 and showed that the elapsed time per modification converges towards approximately 30 milliseconds.

That lazy revisitation becomes cost-efficient already for inter-visitation interval (n) larger than two, and almost independent of transaction size (m) is surprising. However, that the elapsed time converges towards the same value was expected because both approaches have to do almost the same work; for the lazy approach there is an extra overhead in saving the commit times of transactions, administrating which tables should be timestamped, and executing an extra transaction. However, unlike for eager timestamping, more tuples can be timestamped in a single revisit step during lazy timestamping.

8.5 The Cost of Revisiting

We next look at the cost of performing the revisiting step compared to the cost of the actual execution of the transaction. Because neither approach proved superior in the previous study, we look at the revisiting cost for both eager and lazy revisitation.

Figure 13 shows the relative cost of executing the transactions and revisiting the modified tuples for eager revisitation. For small transaction sizes ($m \leq 10$), the revisiting step accounts for 33% to 58% of the total elapsed-time. For larger transaction sizes, the revisiting cost stabilizes at approximately 20% of the total elapsed-time. We have measured the elapsed-time for transaction sizes of up to 1000 modifications. The study shows the revisiting cost to converge towards 17% of the elapsed time.

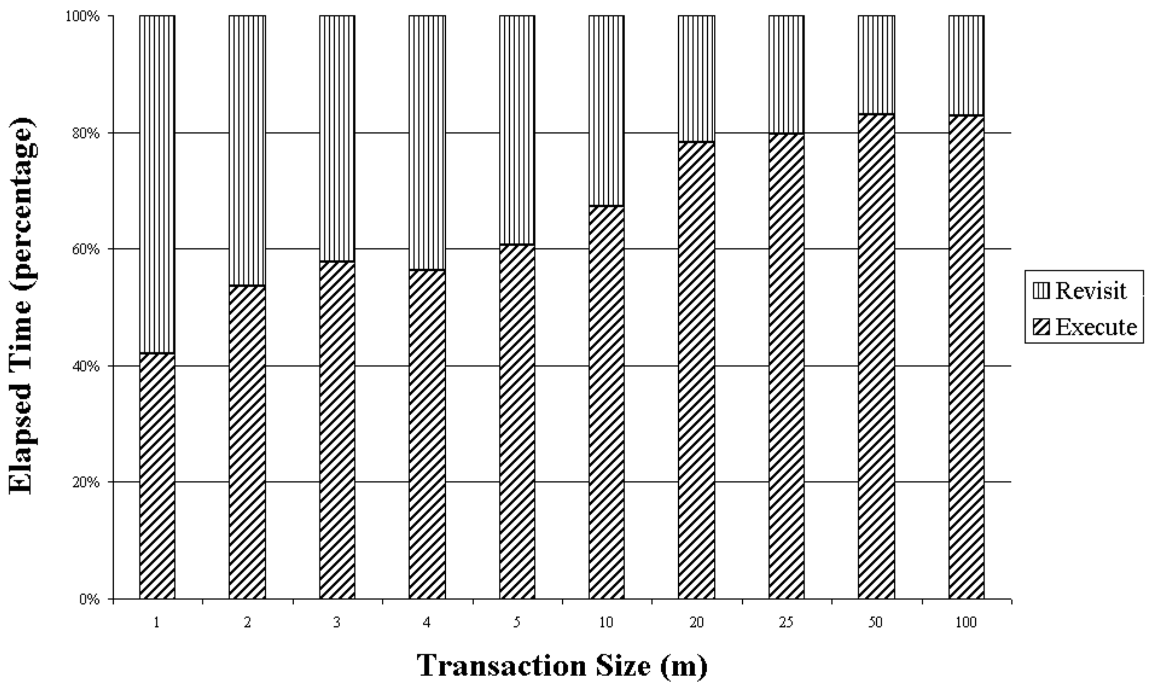


Figure 13: Relative Cost of Transaction Execution Versus Revisiting Using Eager Revisitation

For larger transaction sizes, the overhead of revisiting is very stable because approximately half the tuples to revisit are clustered at the end of the table (the table is stored in transaction-time start order). The other half of tuples to revisit can be found very efficiently using the index on the `TIDS` attribute. The revisiting step is relatively more expensive at smaller transaction sizes because the same number of tuples modified during the actual modification must be revisited, and the clustering of half of the tuples to revisit provides no substantial benefits because only very few tuples are involved.

We now turn to lazy revisitation. Figure 14 shows the relative costs of transaction execution, saving the commit time, and revisiting the modified tuples. We use an inter-visitation interval of five (n=5). Experiments using n=1, n=10, and n=100 (and otherwise identical) showed the same behavior as that reported in Figure 14.

The revisit step in Figure 13 is to be compared with the combination of the save and revisit steps in Figure 14. The total cost of these latter two for the lazy approach is relatively higher than the revisit step for the eager approach because the lazy approaches perform the same tasks as the eager approach, but have extra administration costs for timestamping and use separate transactions for the revisiting.

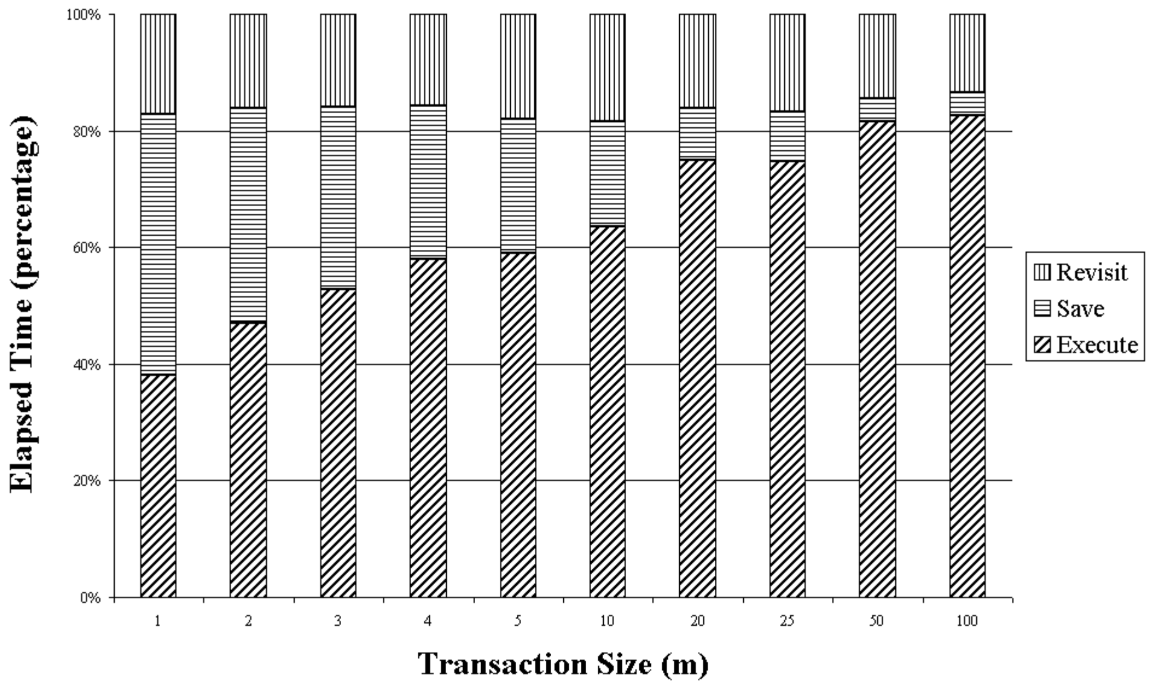


Figure 14: Relative Cost of Transaction Execution Versus Revisiting Using Lazy Revisitation, *n*=5

Figure 14 also shows this administrative cost becomes relatively lower for larger transactions. This is because the save step only stores the TID and timestamp of the transaction, and then commits. The size of this step is independent of the transaction size, making the elapsed time for this step only vary little in absolute numbers.

The relative cost of the revisiting step is constant, at approximately 17%, for various transaction sizes. Although not clear from Figure 14 alone, this indicates that the cost of revisiting grows linearly with the transaction size. The revisiting consists of two parts: (a) timestamping new tuples inserted at the end of the table and (b) timestamping modified tuples that were already present in the table. The first part is nearly independent of the transaction size (or, equivalently, the number of tuples to revisit) because the tuples are clustered on a few disk pages. The cost of the second part grows linearly with the transaction size. The linear growth occurs because the index on the `TIDS` attribute is used to locate the modified tuples. When the tuples are spread evenly over the table, the timestamping of each tuple will consist of an index look-up and the assignment of the permanent timestamp.

Figure 14 indicates that the relative cost of transaction execution versus the revisiting using lazy revisitation is largely independent of the inter-visitation interval (n) and the transaction sizes (m). That is, for varying m, n, and $m \cdot n$, there is a relatively fixed overhead for revisiting tuples. Thus if we can postpone the revisiting step of transactions, e.g., to do it once per night, the number of transactions executed per time unit can be increased by approximately 17% for lazy revisitation. In absolute numbers, this will make lazy revisitation more efficient than eager revisitation, as shown in Figure 15. This figure shows the elapsed-time per modification for eager and lazy revisitation without the revisiting cost. Because the revisiting cost is not included, the time to execute a transaction of size m is independent of n for the lazy approach. This is the reason why the lazy revisitation curves are are almost identical.

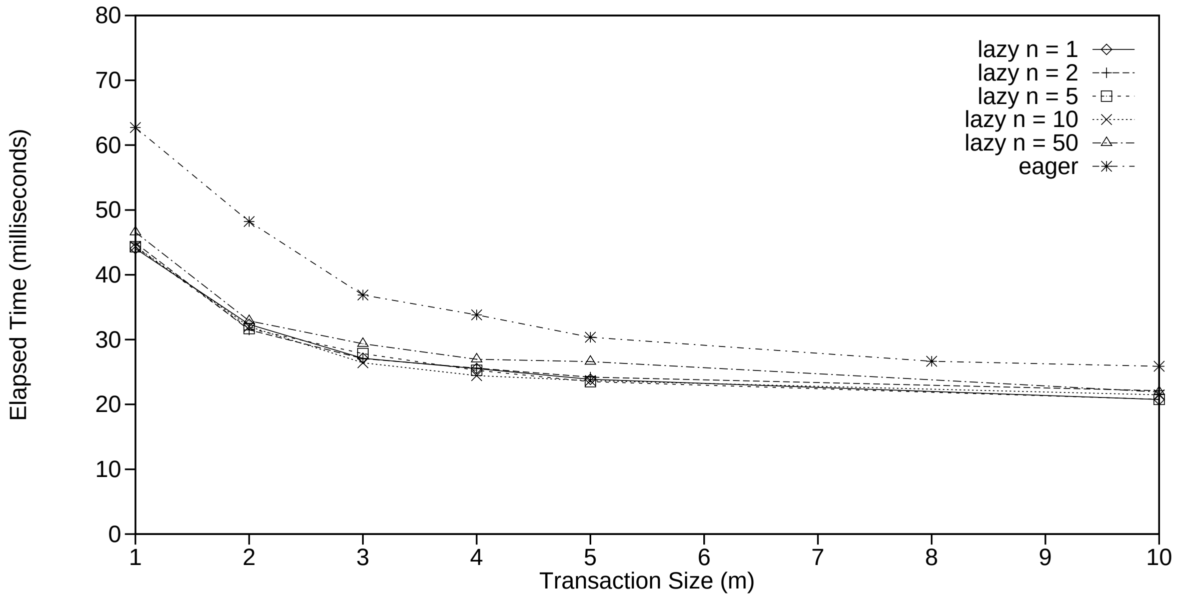


Figure 15: Elapsed Time Per Modification for the Eager and Lazy Approaches Without Revisiting

9 Related Work

Timestamping after commit of time-varying data in local and distributed environments was previously studied by Salzberg [19]. As outlined in Section 4, the present paper extends and refines that study in several respects. While Salzberg is concerned with timestamping the transaction-time dimension, this paper considers also valid time and transaction and valid time together. In Salzberg's study, timeslice queries are considered; this paper proceeds to consider general queries in a temporal SQL. Finally, Salzberg assumes an integrated DBMS architecture, which may may be extended to incorporate a new recovery algorithm and as well as multidimensional temporal indexes; in contrast, this paper describes how timestamping after commit may be achieved in a layer, without necessitating any changes to the underlying DBMS.

Finger and McBrien [10] studied timestamping, including the use of the valid-time variable *now*. They take into consideration that the actual execution of a transaction has a duration in time, and they argue that the value for *now* should remain constant within a transaction. However, they rule out using the commit time for timestamping the valid-time dimension and instead suggest using the start time or the time of the first update for *now*. They showed that using the start time can lead to *now* appearing to be moving backwards in time and—in the case of using the time of the first update—that the serialization of transactions can be violated. They suggest ignoring the problem of time moving backwards or making transactions serializable on their start-times. This paper takes the opposite approach, ruling out using any value for *now* other than the commit time. We show first that the problem of *now* moving backwards cannot be ignored because it may also violate the isolation principle. Second, we argue that transaction executions cannot be serializable in the order of their start times, if concurrency is allowed. Finally, we show that using the commit time, can solve the two problems identified by Finger and McBrien.

An alternative to a stratum approach to building a temporal DBMS is the integrated architecture where the DBMS is built from scratch and the implementation incorporates temporal support. The Postgres DBMS [23, 24] is the best-known system with an integrated architecture. Postgres supports transaction time only and uses timestamping after commit. Commit times of transactions are stored in a special `Time` table. To associate transactions-ids and timestamps with tuples, Postgres adds eight extra attributes to each table. To support both valid time and transaction time, we add five attributes. There is no discussion of temporary values of the timestamp attributes in Postgres. The transaction-time values are left unassigned when a tuple is stored in the database [23]. With respect to revisiting tuples for applying the permanent timestamps, Postgres uses either the “never” or the lazy approach. The integrated architecture of Postgres also permits experimentation with (asynchronous) low-system-usage approach to revisiting tuples [23].

10 Summary and Research Directions

This paper provides a comprehensive approach to timestamping in temporal databases with transaction support as well as support for both the valid-time and transaction-time dimensions.

We show that the straightforward approach to timestamping modifications may lead to violations of the consistency and isolation properties of transactions. To avoid these violations, we formulate a set of requirements for timestamping database modifications. The most important requirements being to preserve of the ACID properties of transaction and to retain a non-reduced level of interleaved transaction execution. The requirements are independent of the underlying temporal database architecture.

For the transaction-time dimension, we use timestamping after commit with revisitation, where permanent timestamps are assigned to the results of the modifications in a transaction only after all statements in the transaction are exhausted and the transaction is ready to commit. The paper provides the details necessary for implementing this timestamping approach in a stratum architecture, where an temporal database management system (DBMS) is built via a layer on top of an existing DBMS. In particular, the paper investigates a spectrum of revisiting strategies, ranging from eager to lazy.

The paper also considers the timestamping of the valid-time dimension. In contrast to previous work, the paper illustrates that the default timestamp values for the valid-time dimension must be identical to values used for the transaction-time dimension, i.e., timestamping after commit must also be used for valid time. This use of timestamping after commit causes problems for the valid-time dimension because of the notion of *now* (the current time) and because users may supply valid times in the future. It is shown that when using the default value for valid-time, isolation level SERIALIZABLE can be obtained; however, for used-supplied valid times in the future, only read-level consistency can be archived.

A performance study demonstrated that for transactions containing few modifications, eager revisitation is the most cost-efficient. For transactions containing more than ten modifications, the eager and lazy approaches are almost equally efficient.

Overall, we have shown how to provide users with simple, consistent, and efficient support for modifying bitemporal databases in the context of user transactions. This can done while fulfilling our requirements, perhaps most notably without lowering the level of concurrency of transactions and without violating the ACID properties.

An interesting topic for future research is the use of more advanced revisitation approaches, e.g., low-system-usage revisitation, in an integrated DBMS architecture. Also the partitioning of the temporal tables, e.g., into old, current, and future data is a topic of future research; partitioning may speed up the revisiting and should be investigated.

References

[1] J. Bair, M. H. Böhlen, C. S. Jensen, and R. T. Snodgrass. *Notions of Upward Compatibility of Temporal Query Languages*. Business Informatics (Wirtschaftsinformatik), 39(1):25–34, February 1997.

[2] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Publishing Company, 1987.

[3] M. H. Böhlen and C. S. Jensen. *A Seamless Integration of Time into SQL*. Technical Report R-96–2049, Aalborg University, Denmark, 1996.

[4] C. J. Bontempo and C. M. Saracco. *Database Management Principles and Products*. Prentice Hall, 1995.

[5] J. Clifford, C. E. Dyreson, T. Isakowitz, C. S. Jensen and R. T. Snodgrass. *On the Semantics of 'Now' in Databases*. ACM Transactions on Database Systems, 22(2):171–214, June 1997.

[6] P. Corrigan and M. Gurry. *Oracle Performance Tuning*. O'Reilly & Associates, 1993.

[7] D. J. DeWitt. *The Wisconsin Benchmark: Past, Present, and Future*. In [11], Chapter 4, pp. 269–315, 1993.

[8] O. Etzion, S. Jajodia, and S. Sripada [eds]. *Temporal Databases: Research and Pratice*. LNCS 1399, Springer Verlag, 1998.

[9] S. Feuerstein. *Oracle PL/SQL Programming*. O'Reilly & Associates, Inc., 1995.

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

[11] J. Gray [ed]. *The Benchmark Handbook for Database and Transaction Processing Systems*. Morgan Kaufmann Publishers, 1993.

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

[13] C. S. Jensen and C. E. Dyreson [eds]. *The Consensus Glossary of Temporal Database Concepts*. In [8], pp. 367–405, 1998.

[14] J. Melton. *Database Language—SQL*, ANSI X3.135-1992, 1992.

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

[16] G. Özsoyoğlu and R. T. Snodgrass. *Temporal and Real-Time Databases: A Survey*. IEEE Transaction on Knowledge and Data Engineering, 7(4):513–532, August 1995.

[17] F. Raab. *Overview of the TPC Benchmark C: A Complex OLTP Benchmark*. In [11] Chapter 3, pp. 131–267, 1993.

[18] M. T. Roth and P. M. Schwarz. *Don't Scrap it, Wrap It! A Wrapper Architecture for Legacy Data Sources*. In Proceedings of the VLDB Conference, Athens, Greece, pp. 265–275, August 1997.

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

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

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

[22] R. T. Snodgrass, M. H. Böhlen, C. S. Jensen and A. Steiner. *Adding Valid Time to SQL/Temporal*. ANSI X3H2-96-501r2, ISO/IEC JTC 1/SC 21/WG 3 DBL-MAD-146r2, November 1996.

[23] M. Stonebraker. *The Design of the Postgres Storage System*. In Proceedings of VLDB Conference, pp. 289–300, 1987.

[24] 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.

[25] K. Torp, C. S. Jensen, and M. Böhlen. *Layered Implementation of Temporal DBMSs—Concepts and Techniques*. Proceeding of the Fifth International Conference On Database Systems for Advanced Applications, pp. 371–380, 1997.

[26] K. Torp, C. S. Jensen, and R. T. Snodgrass. *Stratum Approaches To Temporal Database Implementation*. Proceeding of the International Database Engineering & Application Symposium, pp. 4–13, 1998.

[27] G. Wiederhold. *Mediators in the Architecture of Future Information Systems*. IEEE Computer 25(3):38–49, March 1992.

[28] G. Wiederhold. *Mediation in Information Systems*. ACM Computing Surveys 27(2):265–267, June 1995.

[29] Y. Wu, S. Jajodia, and X. S. Wang. *Temporal Database Bibliography Update* In [8], pp. 338–366, 1998.