

Effective timestamping in databases

Kristian Torp¹, Christian S. Jensen¹, Richard T. Snodgrass²

¹ Department of Computer Science, Aalborg University, Fredrik Bajers Vej 7E, DK-9220 Aalborg Øst, Denmark; e-mail: {torp,csj}@cs.auc.dk

² Department of Computer Science, University of Arizona, Tucson, AZ 85721, USA; e-mail: rts@cs.arizona.edu

Edited by E. Bertino. Received: March 11, 1998 / Accepted July 27, 1999

Abstract. 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.

Key words: Transactions – Timestamping – *NOW* – Valid time – Transaction time

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 [24, 25]. 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 termed bitemporal [14], and this is the type of database considered in this paper.

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 the 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, which 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 system 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 [16]) 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 the 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 Sect. 3. We then outline a new approach in Sect. 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 Sect. 7. A performance study of design alternatives is presented in Sect. 8. Related work is discussed in Sect. 9, and Sect. 10 summarizes and points to directions for future research.

Table 1. The bitemporal table, `Emp`

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

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 closed-open time periods.

A sample bitemporal table is shown in Table 1. The time attributes are called *implicit attributes*, and `Name` and `Dept` are called *explicit attributes*. The implicit attributes capture the time evolution of the table.

The first tuple of Fig. 1 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 later; 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 January 2. 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 third tuple 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.

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 [17, 21, 31]. For example, SQL-92 [16] and SQL3 provide little built-in support, leaving more work to the application programmer. Other languages such as TSQL2 [22] 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 [26, 27]. By adopt-

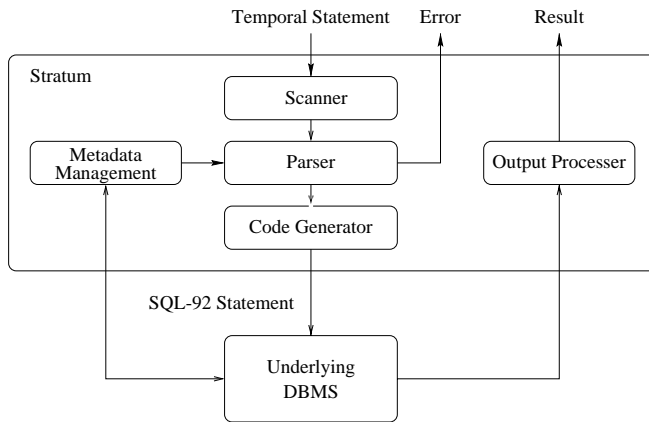


Fig. 1. The stratum architecture

ing a stratum approach, it is possible to maximally reuse existing technology and relatively quickly make a temporal DBMS available to the application programmers, so that they may 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 problems discussed in this paper are relevant for both the stratum and integrated architectures. However, 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 Fig. 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 (or an application accesses the DBMS using an interface such as ODBC [13]). 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.

The stratum approach is similar in some ways to the related area of *mediators* [29, 30] 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* [19]) 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 [23], 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*, *no bind 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* – *no bind 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 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 changed later 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

Table 2. Initial translation of temporal modification statements

Temporal statement	SQL- 92 statement(s)
CREATE TABLE Emp (Name VARCHAR(20), Dept VARCHAR(20)) AS VALIDTIME PERIOD(DATE) AND TRANSACTIONTIME	CREATE TABLE Emp (Name VARCHAR(20), Dept VARCHAR(20), V-Begin DATE, V-End DATE, T-Start DATE, T-End DATE)
INSERT INTO Emp VALUES (<i>new name</i> , <i>new dept</i>)	INSERT INTO Emp VALUES (<i>new name</i> , <i>new dept</i> , <i>now</i> , <i>nobind now</i> , <i>start value</i> , <i>until changed</i>)
VALIDTIME PERIOD [<i>Start</i> - <i>Stop</i>] INSERT INTO Emp VALUES (<i>new name</i> , <i>new dept</i>)	INSERT INTO Emp VALUES (<i>new name</i> , <i>new dept</i> , <i>Start</i> , <i>Stop</i> , <i>start value</i> , <i>until changed</i>)
DELETE FROM Emp WHERE <i>Predicate</i>	INSERT INTO Emp SELECT Name, Dept, V-Begin, <i>now</i> , <i>start value</i> , <i>until changed</i> FROM Emp WHERE <i>Predicate</i> AND T-Stop = <i>until changed</i> AND V-Begin < <i>now</i> AND <i>now</i> < V-End; UPDATE Emp SET T-Stop = <i>stop value</i> WHERE <i>Predicate</i> AND T-Stop = <i>until changed</i> AND V-Begin < <i>now</i> AND <i>now</i> < V-End

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 attributes handled in an ad hoc fashion by applications from time values stored in attributes handled by a temporal DBMS.

The correctness criteria for snapshot transactions running at isolation level SERIALIZABLE [16] 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.

3.2.1 Problems occurring 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 Fig. 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 .

Table 3. Part of table Emp on Day 10 before the update of Jim in T_1

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	<i>until changed</i>
Bob	Toy	1998-01-04	<i>nobind now</i>	1998-01-04	<i>until changed</i>
Jim	Toy	1998-01-04	1998-01-07	1998-01-02	1998- 01-07
Jim	Sports	1998-01-07	<i>nobind now</i>	1998-01-07	<i>until changed</i>

```

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

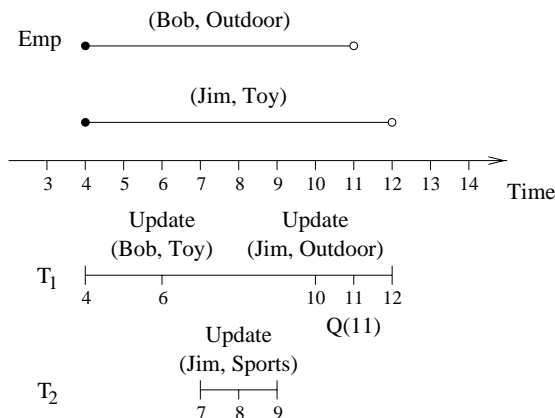
```

Above the time line in Fig. 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 2 days later. Hence, T_2 starts after T_1 starts and commits before T_1 commits.

First consider only transaction T_1 in Fig. 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 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.

Example 1. 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

**Fig. 2.** Problems occurring in temporal transactions

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.

Example 2. As an example using Table 1, the execution of $Q(\text{CURRENT_DATE})$ in a separate transaction on day 13 would not include Jill in its result set; however, the execution of $Q(\text{CURRENT_DATE})$ 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 Fig. 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 non-temporal transactions and queries should return the same results when applied to tables with temporal support as when applied to non-temporal tables.

Example 3. As an example, $Q(11)$ in T_1 will return the two tuples (Bob, Toy) and (Jim, Sports), whereas the equivalent (non-temporal) 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).

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.

Table 4. Part of Table Emp after the Execution of T_1

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	<i>until changed</i>
Bob	Toy	1998-01-04	<i>nobind now</i>	1998-01-04	<i>until changed</i>
Jim	Toy	1998-01-04	1998-01-07	1998-01-02	1998-01- 07
Jim	Sports	1998-01-07	<i>nobind now</i>	1998-01-07	1998-01- 04
Jim	Sports	1998-01-07	1998-01-04	1998-01-04	<i>until changed</i>
Jim	Outdoor	1998-01-04	<i>nobind now</i>	1998-01-04	<i>until changed</i>

The second problem using the start time for *now* is what may be termed history correction, which undermines the accountability of a temporal database.

Example 4. 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 to 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.

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 tuples in Table 4 that shows the result after executing T_1 and T_2 from Fig. 2 with the predicate changed for the valid-time dimension.

Note here that the problems illustrated in the four examples above also occur if we record facts with a granularity of seconds instead of days.

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 Fig. 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 Fig. 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.

3.2.2 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 in Sect. 3.2.1 that the query $Q(\text{CURRENT_DATE})$ 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 transaction¹.

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 2 days to the start time of the transaction and aborting all transactions running for more than 2 days, results in an inaccuracy where a tuple is not visible for query $Q(\text{CURRENT_DATE})$ from the time the transaction actually commits until the time chosen as the commit time of the transaction.

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.” [15, p. 110].

Requirement 5. The result of $Q(t_1)$ at time t_2 should be 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.

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. Otherwise, is the visibility of explicit attributes different in temporal transactions compared to snapshot transactions.

Requirement 7. A transaction must be able to see the values of timestamp attributes of tuples it had previously modified. This is an extension of requirement 6 to include the implicit attributes of a temporal table.

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 [20]. 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 [20].

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 [20]. 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* [20]. 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 mutually 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 Fig. 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 Sect. 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 Sect. 8.4.

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. 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 [20, 24]. 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 Fig. 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

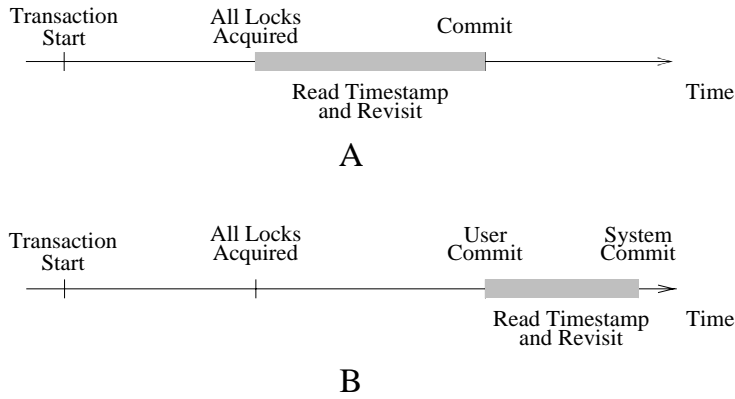


Fig. 3. **A** Temporal transaction. **B** Mapping a temporal transaction to SQL-92

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 Fig. 3B.

Using the approach shown in Fig. 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 that 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 [24]. 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 are discussed in detail in the next two sections. First, we discuss timestamping the transaction-time dimension, then we examine the consequences of introducing 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 Sect. 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 TRANSACTIONTIME2`. As three separate transactions, we issue an insertion, an update, and an update followed by a query, as indicated in Fig. 4. The modifications are expressed in plain SQL-92; the temporal semantics automatically supplies values for the tuple timestamps, as outlined in Sect. 2.

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`.

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.

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

² Again, the details of the temporal extensions are not important. We use a particular syntax [23] only to provide a specific example for expository purposes.


```

-- 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;

```

Fig. 4. Using the transaction-time table, *Emp*

Table 5. The transaction-time table, *Emp*

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	<i>until changed</i>

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 Sect. 8. After cleaning up the *CommitTime* table, the transaction actually commits. If the modification statements in Fig. 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 Fig. 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 multi-user 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 Fig. 4?
- Must modified tuples be revisited before the transaction actually commits?

5.2 Finding a temporary timestamp value

Using timestamping after commit with revisitation, 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 Fig. 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. Note that the time of the first modification within a transaction can be identified by the parser in Fig. 1 and stored as part of the transaction state in the stratum.

We rule out using multiple values of two reasons. First, as discussed in Sect. 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 Sect. 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 transaction 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 [24]. In contrast, storing the transaction id in a transaction-time attribute requires type conversion, because

Table 6. Mapping statements on transaction-time tables into equivalent SQL-92 statements

Temporal statement	Resulting SQL-92 statement(s)
INSERT INTO Emp VALUES (<i>new name</i> , <i>new dept</i>)	INSERT INTO Emp VALUES (<i>new name</i> , <i>new dept</i> , <i>temporary value</i> , <i>until changed</i>)
DELETE FROM Emp WHERE <i>Predicate</i>	UPDATE Emp SET T-Stop = <i>temporary value</i> WHERE <i>Predicate</i> AND T-Stop = <i>until changed</i>
COMMIT	$c \leftarrow \text{CURRENT_DATE};$ INSERT INTO CommitTime VALUES (<i>transaction id</i> , <i>c</i>); UPDATE Emp SET T-Start = <i>c</i> WHERE <i>tuple inserted by transaction id</i> ; UPDATE Emp SET T-Stop = <i>c</i> WHERE <i>tuple deleted by transaction id</i> ; DELETE FROM CommitTime WHERE TID = <i>transaction id</i> ; COMMIT

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 as discussed in Sect. 5.2. The temporary value could, e.g., be stored in a small main memory hash 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` [16]. 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 user-specified 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 eventu-

ally 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 allow (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 Fig. 6. We prefer flexibility in scheduling these database modifications.

Because revisiting tuples adds to the system load, to be discussed further in Sect. 8.4, 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 Sect. 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, 23], 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 `TRANSACTIONTIME SELECT * FROM Emp` in an SQL3 proposal [23]. 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 Sect. 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 [24]; 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 obtain this fine-level degree of control of the underlying DBMS from the stratum.

The “piggy-backing” approach is 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 [24].

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 Sect. 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 [23] `ALTER TABLE Emp ADD VALIDTIME PERIOD (DATE)` on February 1. 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 reflects 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 Fig. 5. On February 1, we insert Kim in the Sports department. On February 2, 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 February 13, we update Kim to be with the Toy department, and on February 16, we delete Kim. Finally, on February 27, 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 implicit 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.

Table 7. The bitemporal table, Emp

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

Table 8. Mapping statements on bitemporal tables into equivalent SQL-92 statements

Temporal statement	SQL-92 statement(s)
INSERT INTO Emp VALUES (<i>new name</i> , <i>new dept</i>)	INSERT INTO Emp VALUES (<i>new name</i> , <i>new dept</i> , <i>now</i> , <i>nobind now</i> , <i>temporary value</i> , <i>until changed</i>)
VALIDTIME PERIOD [<i>Start</i> - <i>Stop</i>] INSERT INTO Emp VALUES (<i>new name</i> , <i>new dept</i>)	INSERT INTO Emp VALUES (<i>new name</i> , <i>new dept</i> , <i>Start</i> , <i>Stop</i> , <i>start value</i> , <i>until changed</i>)
DELETE FROM Emp WHERE <i>Predicate</i>	INSERT INTO Emp SELECT Name, Dept, V-Begin, <i>now</i> , <i>temporary value</i> , <i>until changed</i> FROM Emp WHERE <i>Predicate</i> AND T-Stop = <i>until changed</i> AND V-Begin < <i>now</i> AND <i>now</i> < V-End; UPDATE Emp SET T-Stop = <i>temporary value</i> WHERE <i>Predicate</i> AND T-Stop = <i>until changed</i> AND V-Begin < <i>now</i> AND <i>now</i> < V-End
VALIDTIME PERIOD [<i>Start</i> - <i>Stop</i>] DELETE FROM Emp WHERE <i>Predicate</i>	INSERT INTO Emp SELECT Name, Dept, V-Begin, <i>Start</i> , <i>start value</i> , <i>until changed</i> FROM Emp WHERE <i>Predicate</i> AND T-Stop = <i>until changed</i> AND V-Begin < <i>Start</i> AND <i>Start</i> < V-End; INSERT INTO Emp SELECT <i>A</i> ₁ , ..., <i>A</i> _{<i>n</i>} , <i>Stop</i> , V-End, <i>start value</i> , <i>until changed</i> FROM Emp WHERE <i>Predicate</i> AND T-Stop = <i>until changed</i> AND V-Begin < <i>Stop</i> AND <i>Stop</i> < V-End; UPDATE Emp SET T-Stop = <i>stop value</i> WHERE <i>Predicate</i> AND T-Stop = <i>until changed</i> AND V-Begin < <i>Stop</i> AND <i>Start</i> < V-End

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.

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 February 13, 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 with a user-specified valid-time period followed by an insert with a user-specified valid-time period.

When the user enters commit, 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 Fig. 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

```

-- 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;

```

Fig. 5. Modifying the bitemporal table, Emp

```

-- 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;

```

Fig. 6. A race condition in a delete transaction

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 Fig. 2 again. Assume Jim was inserted in the Shoe department on January 2 and we are using the start time of the transaction for *now*. The result after the execution of T_1 and T_2 is shown in Table 3.

First, notice that the seventh tuple is timestamped with the valid-time period 1998-01-07 – 1998-01-04; the value of $V\text{-Begin}$ is larger than the value of $V\text{-End}$. Second, notice that the fifth and the eighth tuples are both in the current transaction-time state (the current time is 1998-01-04) and have overlapping valid-time periods, namely 1998-01-04 – 1998-01-07 and 1998-01-04 – *now*, respectively, even though Fig. 2 shows that Jim was only in a single department at any point in time.

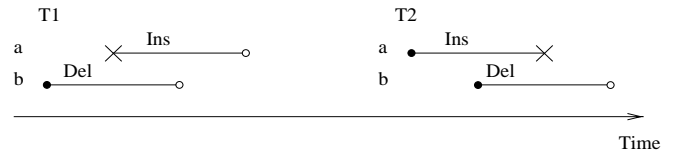


Fig. 7. The general problem using the commit time for *now*

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* in the transaction-time dimension.

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 Fig. 6. This transaction inserts James in the Shoe department and then deletes James from the Sports department in the period 1998-02-01 – 1998-02-21. This leads to a race condition. If the transaction commits on February 20, the delete will have an effect on the inserted tuple. If the clock ticks and the transaction actually commits on February 21, 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 Fig. 7. Here, the circles (filled or non-filled) represent timestamp values given explicitly in a modification statement, and ‘×’ 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 Fig. 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 Fig. 7).

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.

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

Fig. 8. A race condition in an insert transaction

6.3 Handling CURRENT_DATE

The presence of CURRENT_DATE (and CURRENT_TIME and CURRENT_TIMESTAMP) in queries and in modification statements causes some problems. Users expect that this value is identical to the 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.

6.3.1 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 (cf. requirement **R3**) that V-Begin must be smaller than V-End can cause problems in transactions, as shown in Fig. 8. Here we use a period constructor, which takes two SQL-92 date-time expressions as arguments.

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 February 20, no anomalies occur. However, if the transaction does not commit before February 22, 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 Fig. 9, transaction T_1 inserts Joe in the Shoe Department on February 8. On February 19, 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 ?

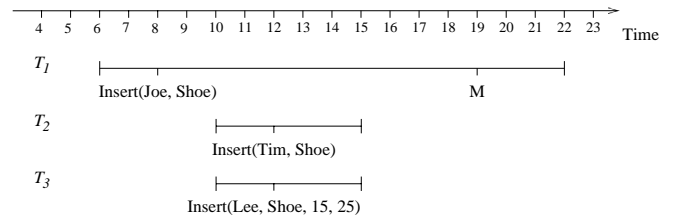


Fig. 9. Using a single value for CURRENT_DATE in queries

The content of the Emp table on February 19, when M is ready to execute, is shown in Table 9. 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.

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.

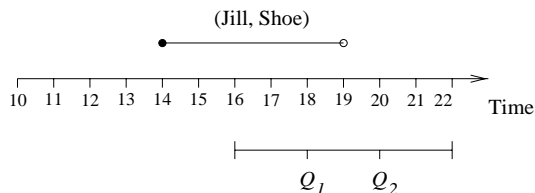
6.3.2 CURRENT_DATE in queries

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 transac-

Table 9. The bitemporal table Emp at 1998-02-19

Name	Dept.	V-Begin	V-End	T-Start	T-Stop
Joe	Shoe	1998-02-08	<i>nobind now</i>	1998-02-08	<i>until changed</i>
Tim	Shoe	1998-02-15	<i>nobind now</i>	1998-02-15	<i>until changed</i>
Lee	Shoe	1998-02-15	1998-02-25	1998-02-15	<i>until changed</i>

**Fig. 10.** CURRENT_DATE in queries

tion, 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 Fig. 10.

This transaction starts on January 16 and commits on January 22. 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 January 18, 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 January 18. On January 20, 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 January 16, and Q_2 will return that Jim is in the Shoe department, even though the query is executed at January 20, 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 Fig. 10 again. This approach evaluates CURRENT_DATE to January 18 and January 20 in queries Q_1 and Q_2 , respectively. This also causes problems. For example, let Q_1 and Q_2 be the same query, retrieving all employees currently in the Shoe department. On January 18, the query will return that Jill is in the Shoe department, and on January 20, the query will find 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 possi-

bility 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 (cf. requirement **R9**). 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 Sect. 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.

Table 10. Representation of the special temporal values

Special value	Representation
<i>nobind now</i>	9999-12-30
<i>beginning</i>	0001-01-01
<i>forever</i>	9999-12-31
<i>until changed</i>	9999-12-31

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 10.

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 $V\text{-End} < '1998-01-10'$ is converted to WHERE CASE WHEN $V\text{-End} = '9999-12-30'$ THEN CURRENT_TIME ELSE $V\text{-End} < '1998-01-10'$. 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 11 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 Sect. 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 Sect. 6.3.

Table 12 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 Sect. 6.3.1. 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 Sect. 8, both DELETE statements in Table 12 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 12.

Finally, Table 13 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 13.

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 13. 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.

The WHERE clauses for the mapping of the COMMIT statements in Table 13 are deliberately vague. This is because two open issues remain: (1) how to associate transaction ids with tuples (discussed in Sect. 5.3), and (2) which of the revisiting approaches identified in Sect. 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 Sect. 5.5 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 Sects. 8.1 and 8.2, we evaluate in Sects. 8.3 and 8.4 the two revisiting approaches anchoring the spectrum, the eager and lazy approaches; both are well suited for implementation in a stratum.

8.1 Objectives of the performance evaluation

With the performance study of revisiting approaches 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?

Table 11. Mapping create table and insert statements on bitemporal tables

Temporal statement	Resulting statement(s)
CREATE TABLE Emp (Name VARCHAR(20), Dept VARCHAR(20)) AS VALIDTIME PERIOD (DATE) AND TRANSACTIONTIME	CREATE TABLE Emp (Name VARCHAR(20), Dept VARCHAR(20), V-Begin DATE, V-End DATE, T-Start DATE, T-Stop DATE)
first modification or use of CURRENT_DATE	<i>temporary value</i> ← CURRENT_DATE
INSERT INTO Emp VALUES (<i>new name</i> , <i>new dept</i>)	INSERT INTO Emp VALUES (<i>new name</i> , <i>new dept</i> , <i>temporary value</i> , <i>nobind now</i> , <i>temporary value</i> , <i>until changed</i>)
VALIDTIME PERIOD [<i>Start</i> - <i>Stop</i>] INSERT INTO Emp VALUES (<i>new name</i> , <i>new dept</i>)	IF (<i>Start</i> is now AND <i>Stop</i> is an explicit timestamp AND <i>smallest.explicit_timestamp</i> > <i>Stop</i>) <i>smallest.explicit_timestamp</i> ← <i>Stop</i> INSERT INTO Emp VALUES (<i>new name</i> , <i>new dept</i> , <i>Start</i> , <i>Stop</i> , <i>temporary value</i> , <i>until changed</i>)

Table 12. Mapping the delete statement on bitemporal tables

Temporal statement	Resulting statement(s)
DELETE FROM Emp WHERE <i>Predicate</i>	INSERT INTO Emp SELECT Name, Dept, V-Begin, <i>temporary value</i> , <i>temporary value</i> , <i>until changed</i> FROM Emp WHERE <i>Predicate</i> AND T-Stop = <i>until changed</i> AND ((V-Begin ≤ <i>temporary value</i> AND <i>temporary value</i> < V-End) OR (V-Begin ≤ T-Start AND V-End = <i>nobind now</i>)); UPDATE Emp SET T-Stop = <i>temporary value</i> WHERE <i>Predicate</i> AND T-Stop = <i>until changed</i> AND ((V-Begin ≤ <i>temporary value</i> AND <i>temporary value</i> < V-End) OR (V-Begin = T-Start AND V-End = <i>nobind now</i>));
VALIDTIME PERIOD [<i>Start</i> - <i>Stop</i>] DELETE FROM Emp WHERE <i>Predicate</i>	IF (<i>Start</i> is now AND <i>Stop</i> is an explicit timestamp AND <i>smallest.explicit_timestamp</i> > <i>Stop</i>) <i>smallest.explicit_timestamp</i> ← <i>Stop</i> INSERT INTO Emp SELECT Name, Dept, V-Begin, <i>Start</i> , <i>temporary value</i> , <i>until changed</i> FROM Emp WHERE <i>Predicate</i> AND T-Stop = <i>until changed</i> AND V-Begin < <i>Start</i> AND <i>Start</i> < V-End; INSERT INTO Emp SELECT Name, Dept, <i>Stop</i> , V-End, <i>temporary value</i> , <i>until changed</i> FROM Emp WHERE <i>Predicate</i> AND T-Stop = <i>until changed</i> AND V-Begin < <i>Stop</i> AND <i>Stop</i> < V-End; UPDATE Emp SET T-Stop = <i>temporary value</i> WHERE <i>Predicate</i> AND T-Stop = <i>until changed</i> AND V-Begin < <i>Stop</i> AND <i>Start</i> < V-End;

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 Sect. 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 Ultra-Sparc-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

Table 13. Mapping start and commit of transactions on bitemporal tables

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

V-BEGIN, V-END, T-START, and T-STOP capture valid and transaction time. Further, the attribute TIDS is added, which encodes the transaction ids of tuples that must be timestamped. A performance study showed that encoding all transaction id in a single attribute was faster than both storing transaction ids in the timestamp attributes or adding a TID attribute for each of the four timestamp attributes. Details can be found in [28].

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. We used a composite B⁺-tree index on the NameId, V-BEGIN, and V-END attributes. This index speeds up modifications and queries. In addition, we add a B⁺-tree index on the TIDS attribute. This index speeds up revisitation.

There are 5000 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 42 MB. Our page size is 8 KB, 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, 11, 18], we only report the elapsed time. 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 Eager versus lazy revisitation

The 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.

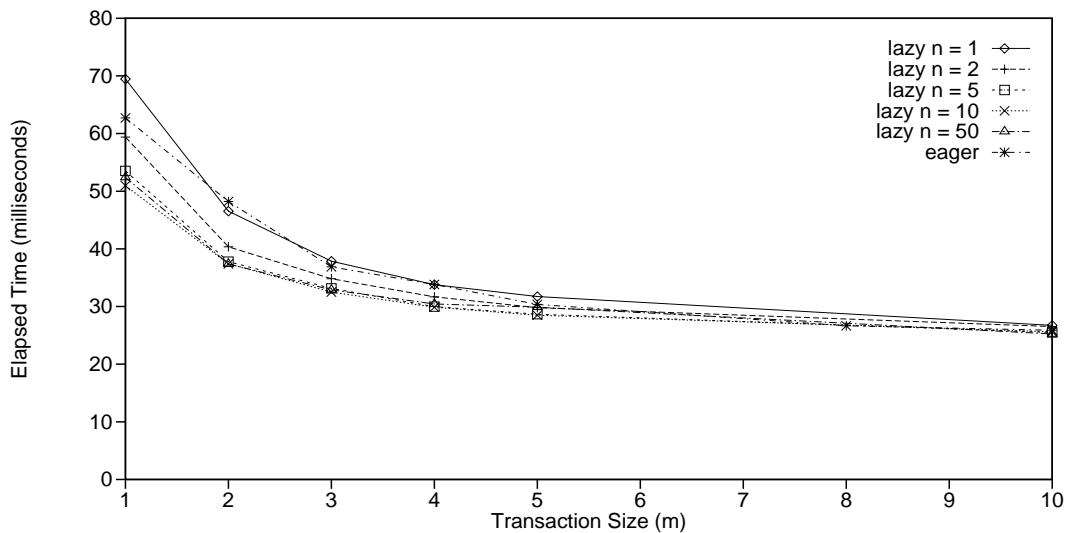


Fig. 11. Elapsed time per modification for eager and lazy revisitation

We use the database schema and associated indexing scheme, discussed in the previous section. We arrived at the indexing scheme after running a set of six test queries and six test modifications on five different indexing schemes and chose the one with the best overall response time. The first index was needed to speed up queries; and the latter index speeds up modifications.

Figure 11 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-visit 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-visit 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 11 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, Fig. 11 shows that the combination of small transactions and lazy revisitation can be expensive on a per-modification basis. This is due to the extra transaction required by lazy revisitation. However, if a transaction contains more than just five modifications, then, independent of the inter-visit interval, the costs of eager and lazy revisitation are virtually identical.

We also ran experiments with inter-visit intervals (n) of up to 200 transactions for transaction sizes (m) of up to 5 modifications; and we experimented with inter-visit intervals of up to 10 transactions for transaction sizes of up to 100 modifications. These experiments were consistent with the trend illustrated by Fig. 11 and showed that the elapsed time per modification converges to approximately 30 ms.

That lazy revisitation becomes cost-efficient already for an inter-visit interval (n) larger than two, and is almost independent of the 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, keeping track of which tables should be timestamped, and executing an extra (revisit) transaction. However, in contrast to eager revisitation, more tuples can be timestamped in a single revisit step during lazy revisitation.

8.4 The cost of revisiting

We next consider the cost of performing the revisiting step as compared to the cost of the actual execution of the transaction. Because neither lazy nor eager revisitation proved superior in the previous study, we study the revisiting cost for both. Figure 12 shows the relative cost of executing the transactions and revisiting the modified tuples for eager and lazy revisitation.

For eager timestamping of small transactions ($m \leq 10$), the revisiting step accounts for between 33% and 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 transactions containing up to 1000 modifications. That study shows that the revisitation cost converges to 17% of the elapsed time.

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 remaining tuples can be found 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 transaction must be revisited, and the clustering provides no substantial benefits because few tuples are involved.

We now turn to lazy revisitation. Figure 12 also shows the relative costs of transaction execution, saving the commit

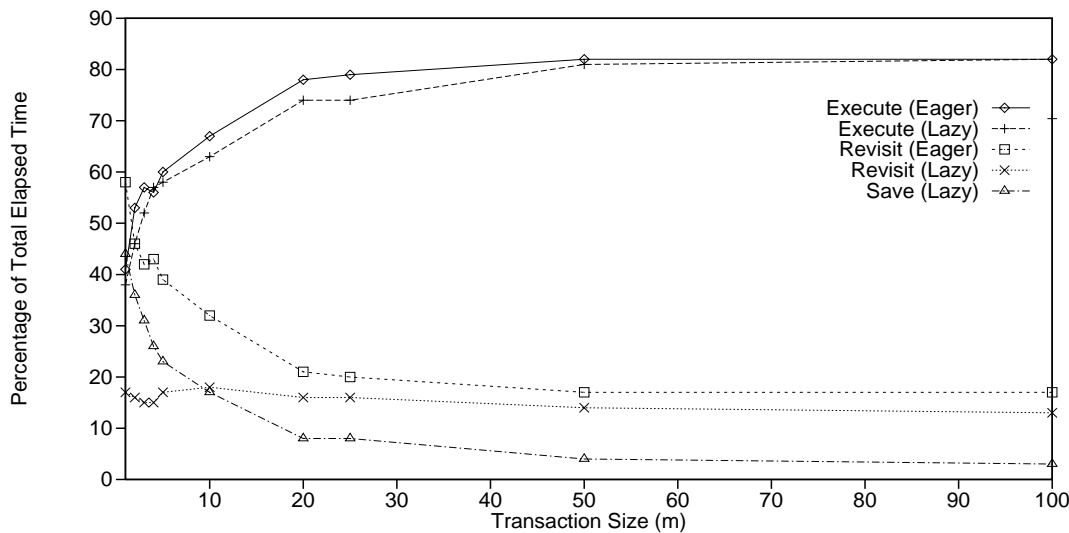


Fig. 12. Relative cost of transaction execution versus revisiting using eager revisitation or using lazy revisitation, $n = 5$

time, and revisiting the modified tuples. We use an inter-visitation interval of $n = 5$. Experiments using $n = 1$, $n = 10$, and $n = 100$ showed similar behavior.

The revisit step for eager revisitation is to be compared with the combination of the save and revisit steps for lazy timestamping. The total cost of these latter two steps 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 an extra administration cost for temporarily recording the timestamp, and because they use separate transactions for the revisiting. Figure 12 also shows that this administrative cost becomes relatively lower for larger transactions. This is because the save step simply stores the TID and timestamp of the transaction, and then commits, which is independent of the transaction size.

The relative cost of the revisiting step is constant, at approximately 17%, for various transaction sizes. Although not clear from Fig. 12 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 12 indicates that the relative cost of transaction execution versus the revisiting using lazy revisitation is largely independent of the transaction size (m). 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 Fig. 13. This figure shows the elapsed time per modifi-

cation for eager and lazy revisitation without the revisiting cost.

9 Related work

Timestamping after commit of time-varying data in local and distributed environments was previously studied by Salzberg [20]. As outlined in Sect. 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 be extended to incorporate a new recovery algorithm and multi-dimensional 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

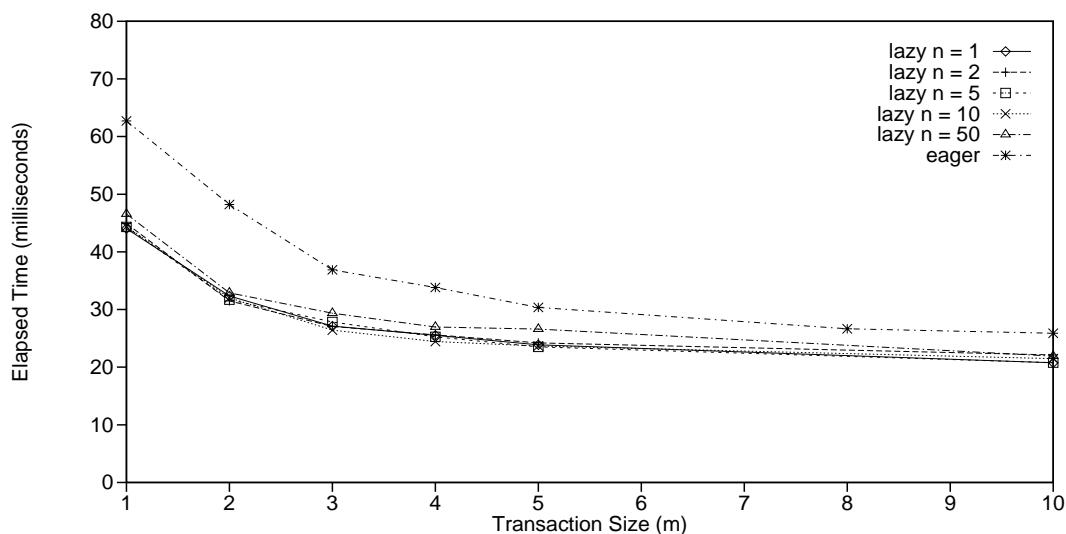


Fig. 13. Elapsed time per modification for the eager and lazy approaches without revisiting

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 [24, 25] 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 [24]. 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 [24].

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 user-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 be 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. Further, the new capabilities in the object-relational DBMSs now being sold, allow parts of the functionality currently implemented in the stratum to be moved closer to the DBMS kernel. It is of interest to determine what functionality may be

moved to the kernel and the performance and other benefits of doing so.

Acknowledgements. This research, conducted in part while the first author visited the University of Arizona, was supported in part by the Danish Technical Research Council through grant 9700780, by the CHOROCHRONOS project, funded by the European Commission DG XII Science, Research and Development, contract no. FMRX-CT96-0056, by a grant from the Nykredit corporation, and by grants IRI-9632569 and IIS-9817798 from the U.S. National Science Foundation. We also thank the reviewers for their helpful comments.

References

1. Bair J, Böhlen MH, Jensen CS, Snodgrass RT (1997) *Notions of Upward Compatibility of Temporal Query Languages*. Business Informatics (Wirtschaftsinformatik) 39(1):25–34
2. Bernstein PA, Hadzilacos V, Goodman N (1987) *Concurrency Control and Recovery in Database Systems*. Addison Wesley, Cambridge, Mass.
3. Böhlen MH, Jensen CS (1996) *A Seamless Integration of Time into SQL*. Technical Report R-96–2049, Aalborg University, Denmark
4. Bontempo CJ, Saracco CM (1995) *Database Management Principles and Products*. Prentice Hall, Englewood Cliffs, N.J.
5. Clifford J, Dyreson CE, Isakowitz T, Jensen CS, Snodgrass RT (1997) *On the Semantics of ‘Now’ in Databases*. ACM Trans Database Syst 22(2):171–214
6. Corrigan P, Gurry M (1993) *Oracle Performance Tuning*. O’Reilly & Associates, Sebastopol, Calif.
7. DeWitt DJ (1993) *The Wisconsin Benchmark: Past, Present, and Future*. In [11], Chapter 4, pp 269–315
8. Etzion O, Jajodia S, Sripada S (eds) (1998) *Temporal Databases: Research and Practice*. LNCS 1399, Springer, Berlin Heidelberg New York
9. Feuerstein S (1995) *Oracle PL/SQL Programming*. O’Reilly & Associates, Sebastopol, Calif.
10. Finger M, McBrien P (1996) *On the Semantics of ‘Current-Time’ in Temporal Databases*. In: Vieira MTP, Traina AJM (eds) 11th Brazilian Symposium on Databases, São Carlos, Brazilian Computer Society, Porto Alegre, Brazil, pp 324–337
11. Gray J (1993) *The Benchmark Handbook for Database and Transaction Processing Systems*. Morgan Kaufmann, San Mateo, Calif.
12. Gray J, Reuter A (1993) *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann
13. Geiger K (1995) *Inside ODBC*. Microsoft Press, Redmond, Wash.
14. Jensen CS, Dyreson CE (eds) (1998) *The Consensus Glossary of Temporal Database Concepts*. In [8], pp 367–405
15. Melton J (1992) *Database Language – SQL*, ANSI X3.135-1992
16. Melton J, Simon AR (1993) *Understanding the New SQL: A Complete Guide*. Morgan Kaufmann, San Mateo, Calif.
17. Özsoyoğlu G, Snodgrass RT (1995) *Temporal and Real-Time Databases: A Survey*. IEEE Trans Knowl Data Eng 7(4):513–532
18. Raab F (1993) *Overview of the TPC Benchmark C: A Complex OLTP Benchmark*. In [11] Chapter 3, pp 131–267
19. Roth MT, Schwarz PM (1997) *Don’t Scrap it, Wrap It! A Wrapper Architecture for Legacy Data Sources*. In: Jarke M, Carey JM, Dittrich KR et al. (eds) Proceedings of the VLDB Conference, Athens, Greece. Morgan Kaufmann, San Mateo, Calif., pp 265–275
20. Salzberg B (1994) *Timestamping After Commit*. In Proceedings of the Conference on Parallel and Distributed Information Systems, Austin, Texas. IEEE Computer Society, pp 160–167
21. Snodgrass RT (1987) *The Temporal Query Language TQuel*. ACM Trans Database Syst 12(2):247–298
22. Snodgrass RT (ed) (1995) *The TSQL2 Temporal Query Language*. Kluwer Academic, Amsterdam
23. Snodgrass RT, Böhlen MH, Jensen CS, Steiner A (1996) *Adding Valid Time to SQL/Temporal*. ANSI X3H2-96-501r2, ISO/IEC JTC 1/SC 21/WG 3 DBL-MAD-146r2
24. Stonebraker M (1987) *The Design of the Postgres Storage System*. In: Stocker PM, Kent W, Hammersley P (eds) Proceedings of VLDB Conference, Brighton, England. Morgan Kaufmann, San Mateo, Calif., pp 289–300
25. Stonebraker M, Rowe LA, Hirohama M (1990) *The Implementation of Postgres*. IEEE Trans Knowl Data Eng 2(1):125–142
26. Torp K, Jensen CS, Böhlen M (1997) *Layered Implementation of Temporal DBMSs – Concepts and Techniques*. In: Topor RW, Tanaka K (eds) Proceedings of the Fifth International Conference On Database Systems for Advanced Applications, Melbourne, Australia. World Scientific Press, Singapore, pp 371–380
27. Torp K, Jensen CS, Snodgrass RT (1998) *Stratum Approaches To Temporal Database Implementation*. In: Eaglestone B, Desai BC, Shao J (eds) Proceedings of the International Database Engineering & Application Symposium, Cardiff, Wales, UK. IEEE Computer Science, pp 4–13
28. Torp K, Jensen CS, Snodgrass RT (1998) *Effective Timestamping in Databases*. TimeCenter Technical Report TR-4, Aalborg University, Denmark
29. Wiederhold G (1992) *Mediators in the Architecture of Future Information Systems*. IEEE Computer 25(3):38–49
30. Wiederhold G (1995) *Mediation in Information Systems*. ACM Comp Surv 27(2):265–267
31. Wu Y, Jajodia S, Wang XS (1998) *Temporal Database Bibliography Update*. In [8], pp 338–366