Correct and Efficient Timestamping of Temporal Data

Kristian Torp, Richard Snodgrass, and Christian S. Jensen

March 11, 1997

TR-4

A TIMECENTER Technical Report

Title	Correct and Efficient Timestamping of Temporal Data
Author(s)	Kristian Torp, Richard Snodgrass, and Christian S. Jensen
Publication History	March 1997. A TIMECENTER Technical Report.

TIMECENTER Participants

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

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

Individual participants
Curtis E. Dyreson, James Cook University, Australia
Michael D. Soo, University of South Florida, USA
Andreas Steiner, ETH Zurich, Switzerland
Jef Wijsen, Vrije Universiteit Brussel, Belgium

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

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

Abstract

Previous approaches to timestamping temporal data have implicitly assumed that transactions have no duration. In this paper we identify several situations where a sequence of operations over time within a single transaction can violate ACID properties.

It has been previously shown that the transaction-time dimension must be timestamped after commit. This time is not known within the transaction. We describe how to correctly implement most queries that make explicit reference to this (unknown) transaction time, and argue that the rest, which can be syntactically identified, can only be answered with an approximation of the correct value.

The drawback of timestamping after commit is that it requires revisiting tuples. We show that this expensive revisiting step is required only before any queries or modifications in subsequent transactions that access prior states; in most cases, revisiting tuples can be postponed, and when to revisit can be syntactically determined. We propose several strategies for revisiting tuples, and we empirically evaluate these strategies in order to determine under which circumstances each is best.

1 Introduction

Temporal database management systems extend conventional database management systems (DBMS's) by providing built-in support for modifying and querying time-varying data [11], thus moving functionality from the applications into the DBMS. Temporal databases supporting transaction time store multiple versions of data by associating time periods with the tuples, thus indicating when they were logically in the database. These time periods contain the special temporal value *until changed*, which denotes the current time.

To get a transaction-consistent view of database modifications, the values given to *until changed* in modifications must be the commit-time of the containing transactions [9]. This causes a problem when data is modified and the modified data then queried in the same transaction. The query can then ask for its containing transaction's commit time, which is not known. The problem is exemplified as follows. The tuple in the temporal `Emp` table in Figure 1 indicates that the tuple (Joe, Shoe) was inserted by some transaction which committed on January 17, 1990 and then logically deleted by a transaction, T, that has (logically) inserted its commit time into the `T-Stop` attribute. The problem is that it is possible later in transaction T to query `Emp` for the transaction time of its tuples. Because the stop time is not known until transaction T commits, the accurate result cannot be returned.

Name	Dept	T-Start	T-Stop
Joe	Shoe	1990-01-17	commit time of T

Figure 1: In Transaction T, What is the Transaction-time Period of the Tuple?

Previous works [4, 8, 9, 10] on timestamping temporal data have either made the implicit assumption that transactions have no duration, or they have not considered transactions that modify and subsequently query the modified data.

In this paper, we examine the problem of modifying and querying time-varying data within the same transaction. We list a set of requirements and goals to render the support for transactions in temporal databases a simple, clear, and consistent extension of the transaction support in conventional databases. More specifically, we identify four subtle problems with can lead to violations of the ACID properties if the value of *until changed* is not chosen properly, and we explore in detail how to assign transaction times to the tuples. We show that in some situation, queries cannot always can return correct results, we identify the queries and situations for which the problems occur, and we suggest different solutions.

Two orthogonal temporal aspects of database facts have been identified. Valid time records when facts are true in the modeled reality, and transaction time records when facts are stored in the database. We focus on the transaction-time aspect, which in itself presents substantial challenges.

The paper is organized as follows. Section 2 describes the database architecture used. Design requirements, design goals, and problems are listed in Section 3. Sections 4–6 provide the details of how to effect correct transaction-time timestamping and present different approaches for timestamping transaction time. A performance study of the design alternatives follows in Section 7. Section 8 discusses related work, and Section 9 concludes and points to future research.

2 The Stratum Approach

How to effect timestamping depends on which underlying architecture is assumed. Three possible architectures are of interest. First, if a conventional DBMS is used, timestamping is the responsibility of the application programmer and is done in the application code. Second, an integrated temporal DBMS architecture may be assumed (as in, e.g., [10]). This architecture permits the DBMS implementor maximum flexibility. Third, a stratum architecture for implementing a temporal DBMS may be assumed. We focus on this latter approach, which is illustrated in Figure 2 and explained next. The downward arrows denote flow of queries, the upwards arrows denote flow of data, and the boxes are software components.

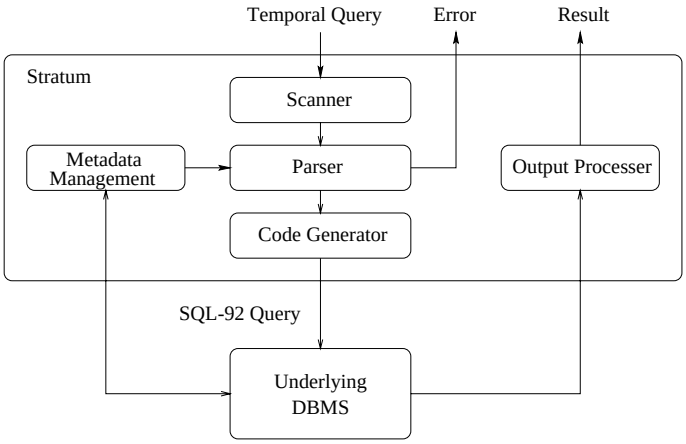

Figure 2: The Stratum Approach

The user enters a temporal query, Q. The stratum converts the temporal query to an SQL-92 query, $\hat{Q}$, which is executed in the underlying conventional DBMS. The DBMS sends the result back to the stratum, which then displays the result of query Q to the user. The user cannot see that the data is actually stored in a conventional DBMS because the stratum totally encapsulates the DBMS from the user's point of view.

We are aware of the primary drawback of using this approach, i.e., well-known temporal storage structures such as temporal indices and temporal join, coalescing, and timeslicing algorithms cannot be used. However, building an integrated temporal database system is a costly task which only the major database vendors can accomplish (to be cost-effective, some vendors even enhance their own systems using a stratum approach). By using a stratum approach, it is possible to maximally reuse existing technology and make a temporal database system available to the users so they can start to take advantages of the new facilities of a temporal extended query language.

We use the temporal query language ATSQL [2] in this paper. This language divides queries on temporal tables into three categories [1]. *Temporal upward compatible* (TUC) ATSQL queries are "regular" SQL-92

[7] queries on temporal tables; they simply consider only the current tuples and thus return a conventional snapshot table. This permits legacy queries to work unchanged when the underlying tables are made temporal. *Sequenced queries* use new temporal features in querying databases in a state-by-state fashion. The result returned by a sequenced query is logically the union of the results of executing a corresponding SQL-92 query on each database state. Finally, *non-sequenced queries* also use temporal features. They have no corresponding SQL-92 counterpart and do not rely on the DBMS to logically split the temporal table into states, as sequenced queries do. Non-sequenced queries are generally used to compare database states.

3 Design Requirements, Goals and Problems

We first list the design requirements and goals for timestamping time-varying data. We then use an example to illustrate four subtle problems which can occur when managing time-varying data.

We assume that all transactions run at the isolation level `SERIALIZABLE` [7] and the underlying DBMS at this isolation level satisfies the ACID properties. Our requirements, on which we do not compromise, are as follows.

- Retain the ACID properties of SQL-92 transactions when temporal support is added.
- Retain the semantics of SQL-92 modifications in transactions, e.g., a deleted tuple is not returned by a subsequent query in that transaction.
- Ensure ACID properties of temporal transactions.
- Retain all the constructs of ATSQL. The implementation techniques should not constrain the functionality of the language.

Our goals, on which some flexibility is allowed, are as follows.

- The effect of temporal modifications within transactions, where the database may be in a temporarily inconsistent state, should be easily understandable; the values of timestamps should be defined by few and simple rules.
- The level of concurrency among transactions should not be lowered considerably when temporal-transaction support is added, e.g., by requiring that all transactions be executed serially.
- The design should be implementable in a stratum approach. This particular goal must be satisfied by our techniques.

The requirements must be fulfilled for the timestamping to be correct. The requirements in combination with the first goal on temporal modifications ensure that temporal transactions are a simple, clear, and consistent extension of SQL-92 transactions. The second goal helps ensure that the design is efficient. The final goal ensures that the approach can be added to a conventional DBMS without necessitating internal changes, and makes it possible for us to evaluate the design alternatives experimentally.

The atomicity property of transactions requires that a state transition is to be executed without any observable intermediate states [5, page 166]. In contrast, the actual execution of a transaction proceeds over a duration in time. Two features in a temporal query language can cause problems[1]. First, there is a notion of current time or *until changed* (the term *NOW* is also used for this concept), which is used extensively both as a timestamp on tuples and as a value in queries. Second, with a temporal query language it is

[1]The first problem also exists in SQL-92. However, the problem is more prevalent in a temporal query language.

possible to query timestamps of data that are not known until the containing transaction commits (explained in Section 4).

The problems that can occur are illustrated in Figure 3. For illustrative purposes, these transactions are shown to require several days in January 1996 to execute. However, the problems occur identically when transactions take only a few minutes, or a few seconds, to execute. Two transactions, T_1 and T_2, are shown. On the dates 1996-01-08 and 1996-01-10, T_1 inserts that Joe is in the Shoe department and that Jim is in the Outdoor department, respectively. On the dates 1996-01-13 and 1996-01-15, T_1 executes the query Q. Transaction T_2 starts after T_1 has committed and also executes the query Q, on 1996-01-20.

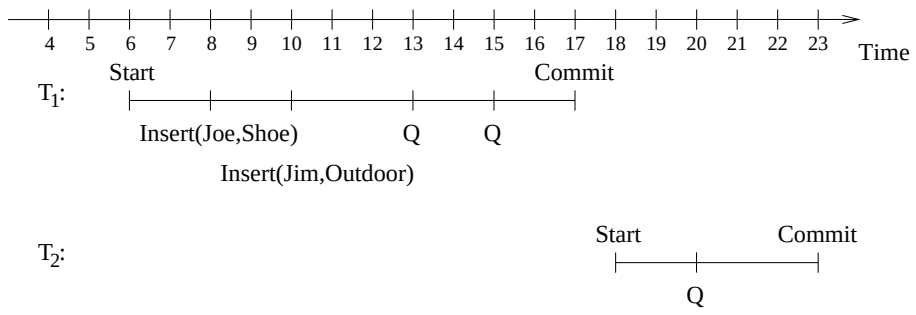

Figure 3: Temporal Modifications and Queries

We want to avoid the following four problems.

1. The fact that the actual execution of transactions has a duration in time must not be detectable by any query. In the example, the query Q, in T_1 or T_2, must not be able to detect that Joe was actually inserted before Jim, because both tuples were inserted by the same transaction. Being able to detect that T_1 actual inserted Joe two days before inserting Jim would violate the atomicity of the transaction.

2. A query, executed twice in a transaction, with no intermediate modifications, must not return different results. In Figure 3, the execution of Q at 1996-01-13 must return the same result set as when Q is executed at 1996-01-15, even when it, e.g., asks for the transaction-time period of the tuple (Joe, Shoe). This may seem counter-intuitive because there are two days between the executions of Q. However, not returning the same result would be similar to a non-repeatable read [5], which violates the isolation of transactions.

3. A query executed in one transaction must not return a result different from the same query executed in another transaction running immediately after it, with no intermediate transactions. In Figure 3, this means that Q must return the same result executed in T_1 or T_2, even when Q refers to the timestamps. Not returning the same result is not easily explainable to application programmers.

4. Timestamps that eventually are identical must not appear temporarily to be different. Not using identical timestamps can cause the wrong, or hard-to-understand, result set to be returned when timestamps are compared.

A first step in preventing these problems is to be able to detect when they may occur.

4 Overall Approach to Transaction Timestamping

We now describe how correct transaction timestamping can be achieved. When one of several alternative approaches is clearly superior, we identify and adopt that approach. When there is no clear winner, we list

the alternatives and postpone choosing one approach until Section 6, where we motivate an overall approach to timestamping.

The requirements discussed in the previous section have two consequences for a transaction timestamping approach. First, it must retain a transaction-consistent view of previous database states. Second, transactions must to be allowed to see their own modifications. We discuss how the consequences shape the overall timestamping approach, and how timestamping can be implemented in a stratum.

To achieve a transaction-consistent view of previous database states, it is necessary to use the same timestamp for all modifications within a transaction [9]. Otherwise, it would be possible to rollback to a time between the times when two modifications occurred in a committed transaction; the committed transaction would then not appear as an atomic action. 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 [9].

To make it possible for transactions to see their own modifications, it may be necessary to associate timestamps with tuples before all locks have been acquired [9]. 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. This is difficult because the correct timestamp is not known before all locks are acquired. The problem can be solved by using a temporary value and then revisit tuples after all locks have been acquired to replace the temporal value with the (now-known) timestamp.

In Figure 4A, we show the times when a transaction starts, when it has acquired all locks, and when the user enters commit. 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, it is not known that all locks have been acquired until when the transaction's final statement is reached, i.e., at the user-commit. Further, a stratum has no access to the internals of the underlying DBMS. We therefore postpone reading the timestamp until after user-commit and then revisit the tuples modified by the transaction, to apply the correct, permanent timestamp; the transaction then actually commits by having the stratum issue a commit to the underlying DBMS. This sequence of events is illustrated in Figure 4B. The details of how to implement this *timestamping after commit* approach in a stratum is discussed next.

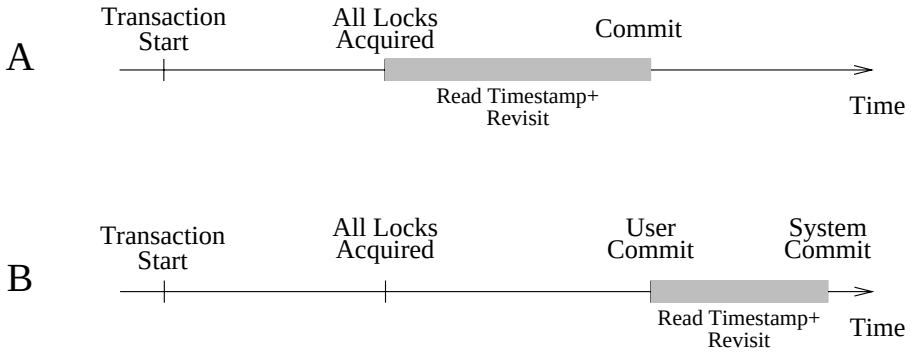

Figure 4: A: ATSQL Transaction B: Resulting SQL-92 Transaction

5 Timestamping After Commit

In this section we first describe the general idea of timestamping after commit using a stratum approach. We give an example which raises four questions. The questions are discussed in turn, thus providing the details of how to implement timestamping after commit.

5.1 General Approach

We use the approach shown in Figure 4B and read the timestamp after user-commit, referred to as the *commit time*. Tuples inserted or updated in a temporal database are assigned the logical transaction timestamp [commit time - *until changed*), where *until changed* means the tuple is recorded as (currently) being in the current state of the database. Tuples logically deleted from a temporal database have their stop value of the transaction timestamp updated to the commit time of the transaction.

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 `Time` table, which has the schema (`TID INTEGER, Commit-Time TIMESTAMP`). We then revisit all tuples modified by the transaction and apply the timestamp stored in the `Time` table. We remove the transaction-ids from tuples and delete the entry in the `Time` table.

As an example, consider the transaction-time table `Emp` from before, which stores the names and departments of employees. To create `Emp` we issue the ATSQL statement `CREATE TABLE Emp (Name VARCHAR(30), Dept VARCHAR(30)) AS TRANSACTION`. As three separate transactions, we issue a TUC-insertion, two TUC-updates, and a sequenced query, as indicated in Figure 5. Note that the modifications are plain SQL-92; the ATSQL semantics automatically supplies values for the tuple timestamps.

```
-- on 1996-01-06:
INSERT INTO Emp VALUES ('Joe', 'Shoe'); COMMIT;
-- on 1996-01-16:
UPDATE Emp SET Dept = 'Sport' WHERE Name = 'Joe'; COMMIT;
-- on 1996-01-27:
UPDATE Emp SET Dept = 'Outdoor' WHERE Name = 'Joe';
TRANSACTION SELECT * FROM Emp;
COMMIT;
```

Figure 5: Using the Transaction-time Table `Emp`

Table 1 shows the `Emp` table after the transactions commit. As can be seen from Table 1 we add two time attributes, `T-Start` and `T-Stop`. The time attributes are called *implicit attributes* and the `Name` and `Dept` are called *explicit attributes*.

Name	Dept	T-Start	T-Stop
Joe	Shoe	1996-01-06	1996-01-16
Joe	Sport	1996-01-16	1996-01-27
Joe	Outdoor	1996-01-27	*until changed*

Table 1: The Transaction-time Table `Emp`

Table 2 shows how ATSQL statements are mapped to SQL-92 statements by the stratum. We assume

that modifications list all explicit attributes. 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 2, is implemented as an ATSQL delete of the old tuple followed by an ATSQL insert of the new tuple. The explicit attributes whose values do not change are copied from the ATSQL deleted tuple to the ATSQL inserted tuple.

When a user enters commit, we record the transaction-id and `CURRENT_TIME` in the `Time` 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. The function *find_time*(transaction-id) returns the commit time of the transaction-id given as an argument by searching the `Time` table. Similarly, tuples deleted by the transaction have their `T-Stop` attribute updated. Having cleaned up the `Time` table, the transaction actually commits. If the modification statements in Figure 5 are translated as indicated in Table 2, we get Table 1.

Table 2 also shows how a TUC-query and a sequenced query are converted from ATSQL to SQL-92. The TUC-query `SELECT * FROM Emp` is converted to a selection of all explicit attributes of tuples currently valid. The (very simple) sequenced query `TRANSACTION SELECT * FROM EMP` is converted to a selection of all explicit attributes, `T-Start`, and `T-Stop` of all tuples in `Emp`.

ATSQL Statement	Resulting SQL-92 Statement(s)
`INSERT INTO Emp VALUES` (A_1 ... A_n)	`INSERT INTO Emp VALUES` (A_1 ... A_n, *temporary value*, *until changed*)
`DELETE FROM Emp` `WHERE` *Predicate*	`UPDATE Emp SET T-Stop =` *temporary value* `WHERE` *Predicate* `AND T-Stop =` *until changed*
`COMMIT`	`INSERT INTO Time VALUES` (transaction-id, `CURRENT_TIME`) `UPDATE Emp SET T-Start =` *find_time*(transaction-id) `WHERE` *tuple inserted by* transaction-id `UPDATE Emp SET T-Stop =` *find_time*(transaction-id) `WHERE` *tuple deleted by* transaction-id `DELETE FROM Time WHERE TID =` transaction-id `COMMIT`
`SELECT * FROM Emp`	`SELECT Name, Dept` `FROM Emp` `WHERE T-Stop =` *until changed*
`TRANSACTION` `SELECT * FROM Emp`	`SELECT T-Start, T-Stop, Name, Dept` `FROM Emp`

Table 2: ATSQL Statements and Equivalent SQL-92-like Statements

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

- How are transaction-ids associated with tuples? 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 correct commit time?
- What is the *temporary value* of the transaction-time attributes for tuples modified within a transaction? As an example, the sequenced selection in Figure 5 is executed before the transaction is committed. The `T-Stop` attribute of the second tuple and the `T-Start` attribute of the third tuple have the temporary value. It is not clear which value should be displayed for these attributes.
- How should *until changed* be represented?

- Must modified tuples be revisited before the transaction commits?

5.2 Associating Transaction-ids With Tuples

We use a transaction-id when revisiting tuples to identify which correct timestamp to associate with the tuples. We see two ways of associating a transaction-id with tuples. First, we may add an extra attribute to each table to store the transaction-id. Second, we may store the transaction-id in the transaction-time attributes.

Using an extra attribute is straightforward: we simply store the transaction-id in this attribute. In contrast, storing the transaction-id in a transaction-time attribute requires type conversion, because the domain of transaction-time attributes is not the same as 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.

Using an extra attribute versus converting transaction-ids to associate a transaction-id within tuples is a trade-off between space and time. The conversion may be useful, but not very elegant in SQL-92 because the conversion between `INTEGER` and `TIMESTAMP` is via an `INTERVAL`. This means we first have to convert a transaction-id to an `INTERVAL` and next add the interval from the smallest value in the time domain. A reverse, two-step approach is needed to decode a transaction-id again.

5.3 Finding a Temporary Timestamp Value

If tuples are to be timestamped with the commit time, tuple modification must be deferred until the transaction commits [9], rendering it impossible for a transaction to see its own modifications. Timestamping tuples with a temporary value before the commit time makes it possible for a transaction to see its own modifications.

Correct transaction timestamps are first applied after user commit, There is therefore a problem when a transaction first modifies the database and then queries it, referring to the transaction timestamps. For example, this is occurs in the last transaction in Figure 5. In general, many queries may refer to the tuples' timestamps in ATSQL. There are several possible responses to this situation.

- We can disallow queries that access the timestamps.
- We can make it a semantic error when a transaction modifies a tuple and subsequently queries the transaction time of that tuple.
- We can warn the user when transaction time is referenced after a modification: the transaction times displayed may change after the transaction commits.
- We can simply return the temporary value stored.

Disallowing references to timestamps restricts the query language, which we will not allow. Simply returning the temporary value is not a clear extension of the transaction semantics. This leaves us with the choice of making it a semantic error or issuing a warning. We find the warning most appropriate because allowing reference to transaction time after modifications within the same transaction is then a transaction design decision.

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 the 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_TIME`.

The first two alternatives will make the modified tuple qualify for the current state and are sensible values to display to the user, along with a warning that the values change when the transaction commits. We rule out using multiple values because it can cause two of the problems discussed in Section 3. First, it 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, we want to avoid temporarily using different values for timestamps that eventually get the same value.

5.4 Representation of *until changed*

A totally separate issue is the representation of *until changed*. All tuples not logically deleted are time-stamped with *until changed* in the `T-Stop` attribute as shown in Table 2. The value for *until changed* cannot be between the time the database was created and the current time, and using a value in the near future is also not a safe option. These representations are either ambiguous or will invalidate the use of a the chosen time value for its normal purpose (i.e., for "representing" itself). Even without these possibilities, several values are still available for representing *until changed*.

- Any time before the database was created.
- The largest value in the domain (usually 9999-12-31).
- 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. This is a complication when we would like to enforce the constraint that the stop value always be after the start value, regardless of whether the stop value is a regular one or is *until changed*. This complication in implementing the stratum can be avoided 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" the `NULL` value without overloading `NULL`. 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 correct, permanent commit times. In Section 4, we described a scenario where the temporary value of the transaction-time attributes is updated to the commit time right after user-commit. Examining which modifications and queries that need to know the correct transaction timestamps, we see that no TUC-modifications and TUC-queries depend on the correct transaction timestamps; rather they simply need to be able to identifying the tuples that are in the current state. Only sequenced and non-sequenced modifications and queries depend on the correct transaction timestamps for their correct execution. As sequenced and non-sequenced queries are syntactically identifiable, syntactic analysis can decide when correct transaction timestamps are required for reasons of correctness of query processing.

Because revisiting tuples adds to the system load, we now explore different approaches for updating the temporary timestamps to the correct commit times, the purpose being to find the most efficient approach. There are several approaches to the revisiting of tuples.

- *Eager*: For each transaction, the correct timestamp is applied immediately, at user-commit.
- *Low-system-usage*: On, e.g., low system load, the tuples are revisited.
- *Piggy-backing*: On pages brought into the buffer, check if any tuples need to be revisited, and do so.
- *Explicitly scheduled revisiting*: Revisit tuples, e.g., at 2 a.m. every night.
- *Lazy*: Revisit only tuples with incorrect timestamps when a query refers to the timestamps and the correct values are needed to process the query correctly.
- *Never*: If a query needs the correct timestamp of a tuple, find it in the `Time` table.

The eager approach was implicitly assumed in Section 5.1. It can be implemented by using after-triggers. The approach is good if timestamps are often referenced in queries and modifications. However, the approach is not cost-efficient if timestamps are rarely referenced.

The "low-system-usage" approach is used in Postgres [10]. However, the approach is not well-suited in a stratum because it requires scheduling of an asynchronous process based on the system load. It is hard to get this fine-level degree of control of the underlying DBMS from the stratum.

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

Explicit scheduling of the revisit is a good choice if TUC-queries exclusively are issued. The approach is not sufficient if there is a mixture of TUC, sequenced, and non-sequenced queries because such queries may not execute correctly if a revisit has not just occurred. It will have to be used in combination with the lazy approach, or the never approach, both of which are described next.

The lazy approach takes advantage of the fact that queries requiring correct 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 exact transaction timestamps for their correctness.

The never approach does not apply the timestamps from the `Time` table to the temporal tables at all, but rather retains the timestamps in a separate table. This requires joining the `Time` table with the temporal table when referring to the transaction-time attributes. This will be expensive for large temporal tables, and only useful if the transaction-time attributes are rarely referenced.

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

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

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

October 1, 1995: `TRANSACTION SELECT * FROM Emp WHERE BEGIN(TRANSACTION(Emp)) >= '1995-01-10'`. For which tuples in `Emp` should the temporal value be replaced with the correct timestamp before evaluating this query?

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

```
NONSEQUENCED TRANSACTION
        SELECT *
        FROM Emp E1, E2
        WHERE BEGIN(TRANSACTION(E1)) > END(TRANSACTION(E2))
```

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

The per-database approach is similar to the per-table approach, except that it brings all tables up-to-date when a query references transaction time. This is also a general approach, but with a more distributed response time than the per-table approach.

6 Picking A Transaction Timestamping Approach

So far, we have explored various alternatives for the ingredients that make up a complete implementation of stratum-based transaction timestamping. Here, we choose a specific composition of the alternatives.

For timestamping the transaction-time dimension we use timestamping after commit. We associate transaction-ids with modified tuples by adding an extra attribute to each temporal table.

For the temporary value of the commit time, we use one value throughout a transaction. Specifically, we chose to use the time of the first modification because this time is that of all constant values available that is closest to the correct, permanent values. We represent *until changed* by the largest value in the domain used for timestamping. We could also use `NULL`. However, this may obviate the use of indexes in many DBMS's.

Two different strategies may be used for revisiting tuples. We can do eager timestamping after each transaction, or we can do lazy timestamping. Because it can be syntactically determined when revisiting is necessary, and because revisiting tuples can be expensive, we will explore the two revisiting strategies in more detail in the following section.

7 Performance Evaluation

Section 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 a DBMS implemented in the stratum. In this section we evaluate the two timestamping approaches delimiting this spectrum, the eager and lazy approaches, both being well-suited for implementation in a stratum approach. We have two goals. First, we want to find out how expensive the revisiting of tuples is in terms of CPU time and disk I/O, compared to the actual execution of the modifications within the transaction. Second, we want to find out under what conditions each timestamping approach is best.

7.1 The Performance Evaluation Setup

We use the Oracle 7.3 DBMS running on a dedicated Digital Alpha server.

While the emphasis has so far been on transaction time, we have previously analyzed TUC-modifications in the valid-time dimension and found that they must be handled similarly to such modifications in the transaction-time dimension. For this reason, we consider here TUC-modifications on the more general bitemporal database case, which supports both valid and transaction time [6].

We only consider TUC-modifications, because we assume this type of modification is the most prevalent in a temporal database system. The modification routines are implemented as stored procedures using the PL/SQL database programming language.

Our test database contains a single bitemporal table `Emp` which has two explicit attributes, `NameId` and `DeptId`, of type `INTEGER`, recording which employees were affiliated with which departments. The four timestamp attributes `V_BEGIN`, `V_END`, `T_START`, and `T_STOP` represent the valid and transaction time dimensions. Each timestamp attribute has an associated TID attribute that specifies which transaction altered the timestamp attribute.

We use nine indices on the table, one on `NameId` and one on each of the timestamp and TID attributes. The indices on `NameId` and the timestamp attributes are used extensively during the actual modification. The indices on the TID attributes are used extensively during revisiting. We found that removing any of the indices negatively impacted performance.

There are 5,000 tuples in the current bitemporal 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 TUC-insert 5%, TUC-delete 5%, and TUC-update 10% of the current bitemporal state. We run our experiments starting with a 18-month old table. This table contains approximately 822,000 tuples, which occupy approximately 30MB. Our page size is 8 KB, and the buffer size of the database is 1.6 MB. Note that this does not imply that the entire current state fit into the buffer because the current state is not stored consecutively on disk.

The tests are performed by executing a total of 2000 TUC-modifications as a series of transactions where we vary the number (m) of TUC-modifications in each transaction; and for the lazy approach, we also vary the number (n) of transactions between revisiting tuples, termed the *intervisitation interval*. The CPU time and number of I/O operations are measured by querying Oracle's dynamic tables 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 CPU time and I/O operations before a revisit. The numbers we report here are the average of our measurements. Before each test run, we scramble the database buffer by scanning a large table. After each test run, we restore the database to the 18-month old state.

7.2 Number of Modifications in Transactions

We first evaluate how the number of modifications m in a transaction affects both the eager and the lazy timestamp approaches. We start with the eager.

Figure 6 shows the CPU time to the left and the number of I/O operations to the right when using the eager timestamping approach. The black and gray columns represent the user phase and commit phase of the transaction, respectively. Recall from Figure 4 that the user phase is the activity in the transaction before the user enters commit and the commit phase is when the transaction reads the commit time and revisits tuples.

As can be seen, both the CPU time and the number of I/O operations increase as the transaction performs more modifications, as expected. The non-linearity for the CPU time for m between 5 and 10 is probably due to vagaries in the buffer management algorithm in Oracle. Note, the CPU time used for the commit phase increases with m whereas the number of I/O operations for commit is almost independent of m.

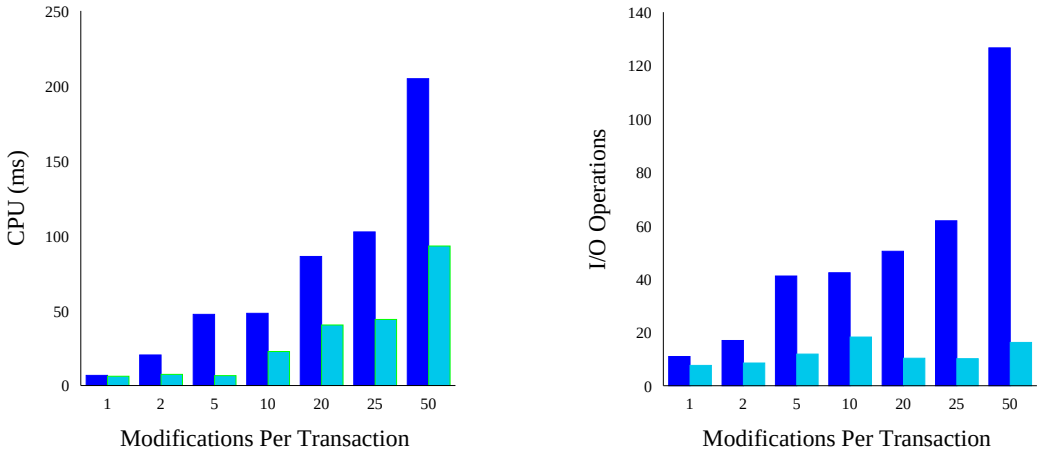

Figure 6: User Phase vs. Commit Phase for Eager Approach

In Figure 7 we have normalized Figure 6 to show the relative size of the user and commit phases. The black part is the user phase and the gray part is the commit phase.

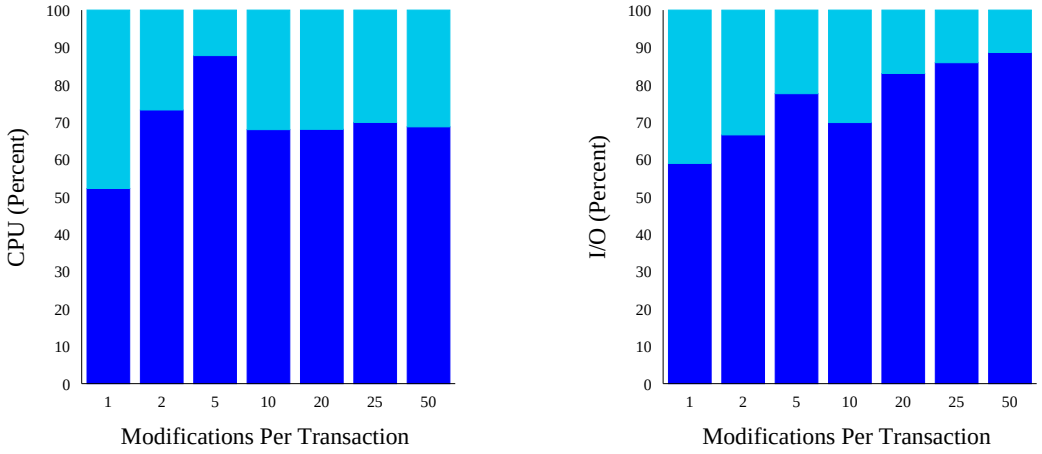

Figure 7: Relative Contribution of Commit Phase for Eager Approach

The percentage of CPU time spent in the commit phase stabilizes around 30% as the number of modifications (m) increases. One might anticipate that it would be closer to 50% because the number of pages being handled in the two phases is approximately the same. We believe the lower percentage is due to the fact that in the user phase, the timestamp attributes appear in comparisons while in the commit phase, the comparison primarily refers to the TID attributes; we found it is faster (2-3 times) to compare integers (the TID attributes) than comparing dates (the timestamp attributes) in Oracle.

The relative part spent in the commit phase for I/O decreases as the transactions do more modifications. It appears that this is due to a buffer effect. Tuples inserted in a transaction will be stored almost consecutively on disk. As a transaction performs more and more modifications the likelihood increases that a single page contains several tuples that must be revisited.

We next turn to examining how the number of modifications affects the lazy timestamping approach. Figure 8 shows the CPU time and the number of I/O operations for the lazy approach when tuples are

revisited for every five transactions. The black part still refers to the user phase. The gray part now indicates both the commit phase and the CPU time or I/O operations needed to revisit (divided by the number n of transactions since the last revisit).

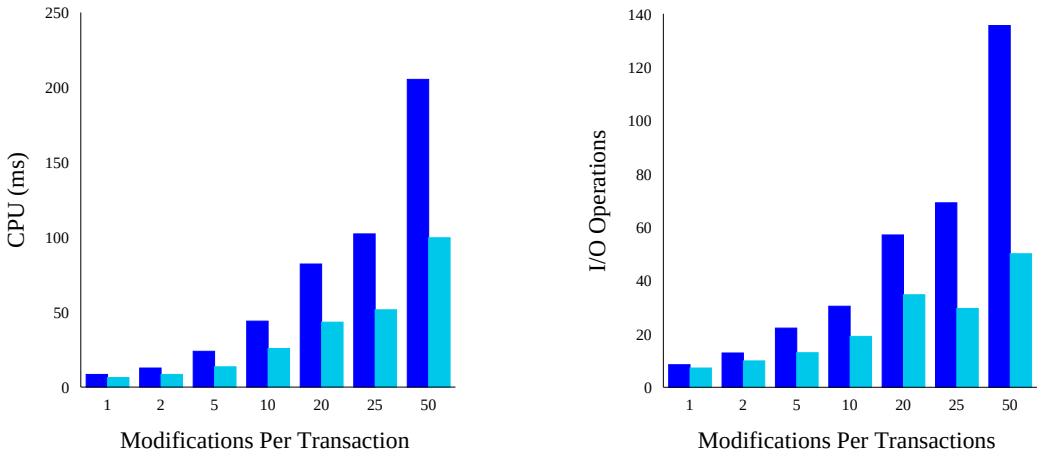

Figure 8: User Phase vs. Commit Phase for Lazy Approach with n = 5

Figure 8 is quite similar to Figure 6 for the user phase. This result was expected because the approaches share a common code base for the procedures executed in the user phase. For the commit phase, the CPU times for the two approaches are also very similar. The number of I/O operations is larger for the lazy approach and increases with m, unlike for the eager approach. This increase in I/O operations was expected because as m increases, pages may be flushed from the buffer and have to be brought into main memory again for revisiting.

In Figure 9 we have normalized Figure 8. The black area indicates the user phase, the dark-gray area indicates the commit phase, and the light-gray area indicates a transaction's portion of the CPU time and I/O operations used for revisiting.

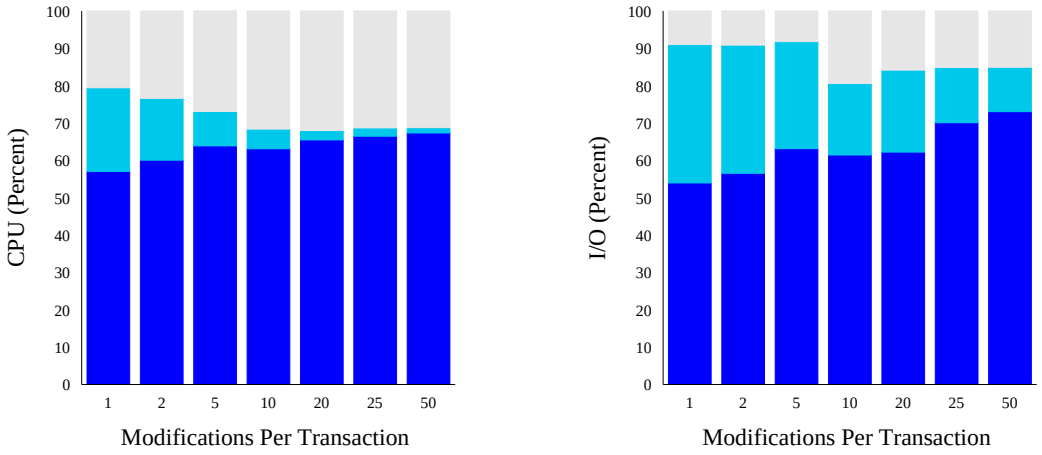

Figure 9: Relative Contributions of User Phase, Commit Phase, and Revisiting for Lazy Approach

For both the CPU time and number of I/O operations the relative part used in the commit phase decreases with m because this phase only stores the commit time of the transaction and then commits. However,

the number of I/O operations needed to commit a transaction remains fairly high even as m increases. Compared to the eager approach, the relative CPU time and number of I/O operations needed to commit and revisit tuples is higher for the lazy approach. It appears that this is because the eager approach has a higher buffer hit rate.

7.3 The Best Timestamping Approach

We now vary both the number of modifications m in a transaction and the number n of transactions between revisits to learn in which situations each of the two timestamping approaches is best.

Figures 10 and 11 show the total CPU time and number of I/O operations on a per-transaction basis for different values of m and n.

With respect to CPU time, the lazy approach is better than the eager approach when the number of modifications m per transaction is somewhere between 2 and 20. This holds regardless of the length of the intervisitation interval. With respect to the number of I/O operations, the lazy approach is better for m between 2 and a value below 20. When n = 20 it is 50% to 100% more expensive to use the lazy approach over the eager approach.

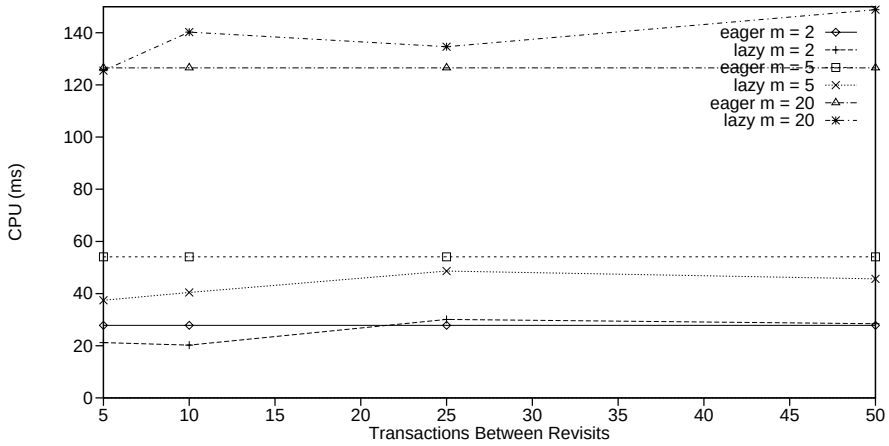

Figure 10: Per-Transaction Total CPU time Cost, Including the Revisit Step

We next turn to look at which timestamping approach is better when we take the cost of revisiting out, thus emphasizing response time instead of throughput. Figures 12 and 13 show the CPU time and number of I/O operations per transaction for different values of m and n without the cost of revisiting.

Considering CPU time, the lazy approach is better for all values of m and n tested. The lazy approach is better because it does less work than the eager approach. For the number of I/O operations, the lazy approach is better when the value of m is between 2 and 5. However, when m = 20, the eager approach is better for $n > 10$. Again, inference with the buffer management strategy in Oracle may be the culprit.

From these studies, we conclude that the best timestamping approach is generally a trade-off between throughput and response time. If the highest throughput is desired, the eager approach should be used because better buffer hit rates lowers its total CPU time and number of I/O operations. If the best response-time is wanted and timestamping can be postponed, e.g., to be done each night, the lazy approach is better because it has a simpler commit phase where only the commit time is stored, compared to the eager approach where all modified tuples are revisited.

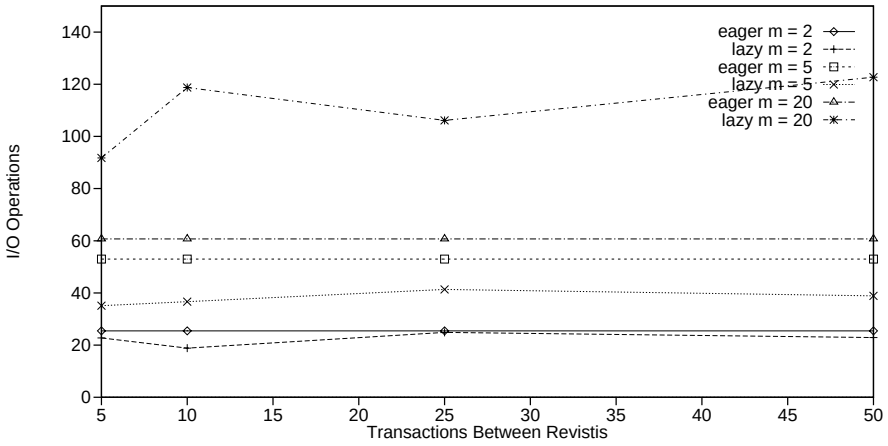

Figure 11: Per-Transaction Total I/O-Cost, Including the Revisit Step

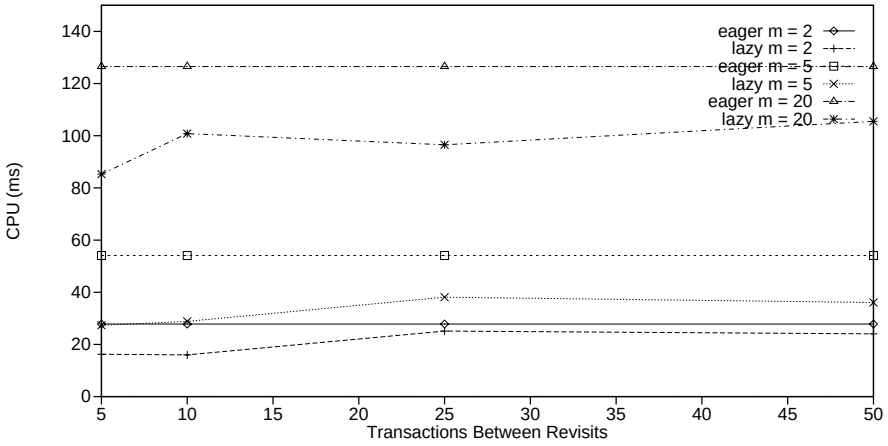

Figure 12: Per-Transaction CPU time Cost, Without the Revisit Step

8 Related Work

Salzberg [9] studied the timestamping of time-varying data in a local and distributed environments, and suggested timestamping after commit, which is also adopted in this paper. She did not discuss transactions containing modifications followed by queries which refer to timestamps.

With respect to revisiting tuples for applying the correct timestamps, Postgres [10, 8] uses the lazy or the never approach in its integrated architecture. The commit times of transactions are stored in a special `Time` table. When a query uses the timestamp attributes, the commit times are retrieved from the `Time` table in the never approach; in the lazy approach, the timestamps are then applied to the timestamp attributes.

Finger and McBrien [4] studied the use of *NOW* in the valid-time dimension, which corresponds to *until changed* for transaction time. They take into consideration that the actual execution of a transaction has a duration in time and also 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 suggest instead using the start time or the time of the first update as the permanent value of *NOW*. They do observe that this can lead to the anomalous behavior of *NOW* seeming to move backwards, and to timestamps

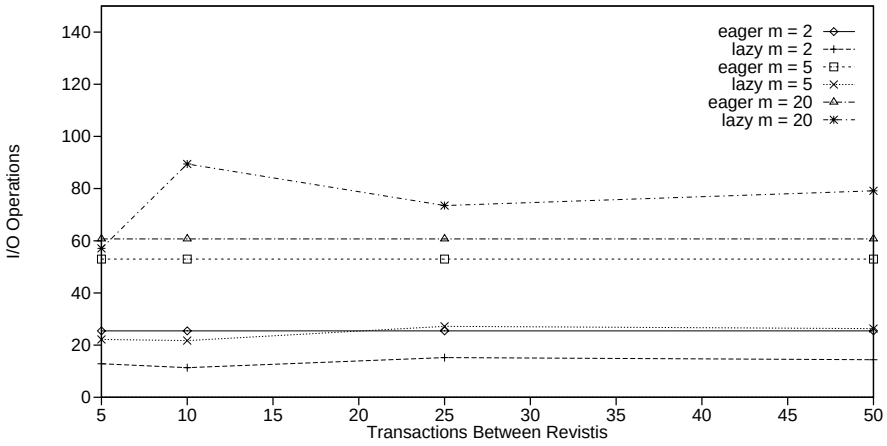

Figure 13: Per-Transaction I/O-Cost, Without the Revisit Step

inconsistent with the serialization order of the transactions.

9 Conclusion and Future Research

The ACID properties of transactions are fundamental to database systems. In this paper we identify situations where the ACID properties can be violated when time-varying data is modified and subsequent queries make reference to the time attributes.

We listed a set of requirements and goals on a timestamping approach, most prominently to avoid violating the ACID properties. We examined in detail how to implement timestamping after commit. In doing so, we studied the conflict between the consequence of our goal of retaining the ACID properties of transactions versus always returning an accurate, or correct, result from a temporal query. The former requires the use of a temporary, approximate value for timestamps that is changed into a correct, permanent values when the transaction commits. The latter may display the temporary value to the user. However, all violations can be syntactically identified and a warning issued to the user.

It is shown that the most commonly used queries and modifications do not depend on the precise timestamps for their correctness. We can exploit this to postpone the revisiting step, which is required by timestamping after commit. We evaluated several alternatives, and showed under which circumstances eager and lazy revisiting strategies are preferred.

In terms of future research, work is needed to determine how to timestamp the valid-time dimension. While the considerations for transaction time apply, this dimension adds the complications that timestamps now may be supplied by the user, not only the system, and that and there are several special temporal values. The effects of buffer size and replacement strategy on the performance of the revisit step also deserve study. Finally, a more detailed performance study taking temporal queries from multiple concurrent users on several tables into consideration would be beneficial, to determine in greater detail when the different revisiting strategies are preferred.

References

[1] J. Bair, M. H. Böhlen, C. S. Jensen, and R. T. Snodgrass. Notions of Upward Compatibility of Temporal Query Languages. *Wirtschaftsinformatik*, to appear, 1997.

[2] M. H. Böhlen and C. S. Jensen. A Seamless Integration of Time into SQL. Technical Report R-96–2049, Aalborg University, Department of Computer Science, Frederik Bajers Vej 7E, DK–9220 Aalborg Øst, Denmark, December 1996.

[3] P. Corrigan and M. Gurry. *Oracle Performance Tuning*. A Nutshell Handbook. O'Reilly & Associates, Sebastopol, CA, 1993.

[4] M. Finger and P. McBrien. On the Semantics of 'current-time' in Temporal Databases. In *Proceedings of the 11th Brazilian Symposium on Databases*, pp. 324–337, October 1996.

[5] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, San Francisco, CA, 1993.

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

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

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

[9] B. Salzberg. Timestamping After Commit. In *Proceedings of the Conference on Parallel and Distributed Information Systems*, September 1994.

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

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