Proceedings of the 27th VLDB Conference, Rome, Italy, pp. 441-450, September 11-14, 2001.

Transaction Timestamping in (Temporal) Databases

Christian S. Jensen

Aalborg University
Fredrik Bajers Vej 7E
DK-9220 Aalborg Øst, Denmark
csj@cs.auc.dk

David B. Lomet

Microsoft Research
One Microsoft Way
Redmond WA 98052, USA
lomet@microsoft.com

Abstract

Many database applications need accountability and trace-ability that necessitate retaining previous database states. For a transaction-time database supporting this, the choice of times used to timestamp database records, to establish when records are or were current, needs to be consistent with a committed transaction serialization order. Previous solutions have chosen timestamps at commit time, selecting a time that agrees with commit order. However, SQL standard databases can require an earlier choice because a statement within a transaction may request "current time." Managing timestamps chosen before a serialization order is established is the challenging problem we solve here.

By building on two-phase locking concurrency control, we can delay a transaction's choice of a timestamp, reducing the chance that transactions may need to be aborted in order keep timestamps consistent with a serialization order. Also, while timestamps stored with records in a transaction-time database make it possible to directly identify write-write and write-read conflicts, handling read-write conflicts requires more. Our simple auxiliary structure conservatively detects read-write conflicts, and hence provides transaction timestamps that are consistent with a serialization order.

1 Introduction

A conventional database relation contains a set of tuples, or records. Insertions, updates, and deletions render this set time-varying. When introducing transaction-time support to such a relation, not only is its evolving current state available, but so also are previously current, and now past states. This type of database support is desirable in applications where accountability or trace-ability are important, which is the case in financial, insurance, and medical applications, to name but a few.

We consider a transaction-time relation as consisting of a set of data items, which may be thought of as records. Each data item d has two system-maintained attributes, denoted by $d.\mathrm{TT}^{\vdash}$ (start time) and $d.\mathrm{TT}^{\dashv}$ (end time). These two values define a closed-open time interval $[d.\mathrm{TT}^{\vdash}, d.\mathrm{TT}^{\dashv})$ during which data item d was part of the current database state. So, the first of these records when data item d became part of the current state, and the latter records when d ceased to be part of the current state.

To obtain this semantics, modification statements update these times as follows. A statement that inserts a data item d sets $d.\mathrm{TT}^{\vdash}$ to the "current time," denoted by $t_{current}$. For now, $t_{current}$ may be thought of as the value of the system clock when current time is referenced, here when the insertion is executed; the remainder of the paper discusses the actual choice of $t_{current}$ in substantial detail. Next, the insertion sets $d.\mathrm{TT}^{\dashv}$ to now, which denotes a variable that evaluates to the current time [2]. The pair of timestamps indicates that d is a current data item and remains so until this is changed explicitly by a delete or update statement. A statement that deletes data item d simply sets $d.\mathrm{TT}^{\dashv}$ to $t_{current}$, indicating that d ceases to be current at time $t_{current}$. Update operations are usually implemented as deletions of the original items to be updated, followed by insertions of the updated items.

A common query, termed a timeslice, is to ask for the set of data items that were current in an argument relation as of some time t not exceeding the current time. This is answered by finding each data item d for which $t \in [d.\mathrm{TT}^{\vdash}, d.\mathrm{TT}^{\dashv})$ (if $d.\mathrm{TT}^{\dashv}$ is now, the current time plus one time unit is used in its place). So the timeslice with time parameter t returns the state that was current at time t.

When user-specified transactions are supported, it is also necessary to use the same $t_{current}$ value for all statements in the same transaction. Otherwise, the atomicity of transactions is compromised—it would be possible for a timeslice to return a transaction internal, and thus inconsistent, database state. Using the same $t_{current}$ for all statements of a transaction makes all actions of a transaction conceptually take place at the same time. This is a necessary refinement of the SQL standard, whose current definition allows different statements in the same transaction to

**Proceedings of the 27th VLDB Conference,
Roma, Italy, 2001**

use separate $t_{current}$ values[1].

The specific choice of current time that is used for timestamping the data items is also essential to ensuring that any previous state that can be retrieved via a timeslice with t as its parameter is indeed the state that was current at time t. If the transaction timestamp order does not agree with a serialization order, it can happen that the answer to the timeslice query never existed as a current state.

In the SQL database language [9], a query or modification can reference current time, $t_{current}$. As described above, current time can be stored as an attribute in the database or used to query the database, e.g., retrieving the state that was current ten minutes ago. Referencing $t_{current}$ in a query can force the database management system (DBMS) to choose this time before the transaction commits. This exposes the transaction to the risk that the $t_{current}$ value given to it and the values given to other transactions are not ordered in a way consistent with a valid transaction serialization order.

time	T_1	T_2
1	fix $t_{current}$	
2	$w((x, 10, [1, now)))$	
3		fix $t_{current}$
4		$w((y, 31, [3, now)))$
5		commit
6		
7	$r((y, 31, [3, now)))$	
8	$w((z, 14, [1, now)))$	
9	commit	

Figure 1: Schedule with Early Choice of Current Time

Figure 1 illustrates the above problem. For simplicity of exposition, we elide the writes done to delete old versions when updates occurs. A read-write conflict on data item y between transactions T_1 and T_2 puts T_2 before T_1 in any serializable schedule involving these two transactions. In addition, T_1 chose its $t_{current}$ value at time 1, in preparation for the write statement at time 2, while T_2 chose its $t_{current}$ value at time 3. Note also that this schedule is allowed by two-phase locking. This results in serialization order being different from timestamp order, and causes two potential problems. First, a timeslice for $\{y, z\}$ for time 2, issued at time 6, returns $\{y = y_0, z = z_0\}$ since T_2 has a time later than time 2 and T_1 has not yet committed or even accessed z yet. However, the same timeslice (i.e., for time 2), instead issued at time 10, returns $\{y = y_0, z = 14\}$ because it sees the z written by T_1. Second, clearly one of these timeslice results (the later one in this case) is not a transaction consistent view of the database. This is unacceptable.

It is easy to avoid the above anomalies if a transaction T's timestamp can be established at the time at which T is committing, as the timestamp can then be chosen to agree with the commit ordering of the transactions [8, 12, 13]. But if one is forced to choose the time at an earlier point in the transaction execution, e.g., when the transaction asks for the current time, keeping the timestamp order consistent with a valid transaction serialization order can become a substantial problem.

Indeed, it is not straightforward how to accomplish timestamping in the face of user transactions and early timestamp choice, while avoiding excessively restricting potential concurrency and excessive transaction aborts. The paper solves this problem. Prior studies of transaction timestamping (covered in some detail in Section 2 and more broadly in Section 5) either did not address early timestamp choice or failed in one of the respects mentioned here.

Our solution, which exploits timestamps, ensures that the order of $t_{current}$ values used by the transactions is consistent with the order of transaction commit. More specifically, the choice of $t_{current}$ in our solution satisfies three requirements. First, if a transaction T has $t_{current}$ = t then T has started and not yet committed at time t. Second, if transaction T_1 has been assigned $t_{current} = t_1$ and transaction T_2 has been assigned $t_{current} = t_2$ and $t_1 < t_2$ then T_1 cannot see data items written by T_2. Third, if the same conditions hold, we require that an equivalent serial schedule exists where T_1 is before T_2.

By ensuring that transaction timestamp order agrees with transaction serialization order, we avoid the two anomalies exposed in Figure 1. The contents of a timeslice cannot change because subsequent writers of data items in a timeslice are required to have later timestamps than the timestamp used by the timeslice, so as to enforce read-write conflict order. Also, because timestamp order agrees with serialization order, each timeslice ”sees” a transaction consistent version of the database.

Our solution does not use timestamp order to serialize transactions, but rather enforces timestamp order as an addition to the two-phase locking normally used to enforce transaction serializability. The solution improves upon prior solutions for the problems that originate when a transaction is assigned a timestamp early in its execution.

1. We choose a transaction timestamp as late as logically possible. We exploit the fact that timestamps are not used to provide concurrency control. Rather, we assume that the database system uses two-phase locking for concurrency control. Our method only needs to keep transaction timestamps consistent with the serialization order resulting from two-phase locking. In particular, we use locking to avoid the need to have a timestamp as of transaction start. Locking will serialize active transactions that do not have a timestamp. We describe how to do this in Section 3.

2. We offer a range of granularities for the bookkeeping of timestamp constraints, hence controlling the trade-

[1] While the standard fixes the value 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.” [9, p. 110].

off between the chances that the constraints cannot be satisfied, leading to abort, and the cost of bookkeeping. It is thus not necessary to provide timestamps for every data item, which has high bookkeeping cost. Rather, we can be conservative to varying degrees in our timestamp testing. Hence, a read timestamp T^{R} needs not be uniquely assigned to exactly one data item, but can apply to some set of data items. While this imprecision increases the probability of abort somewhat, we can control this probability by varying the granularity. In Section 4 we describe our simple technique for this.

It is helpful in our presentation to discuss some of the past solutions right away, in Section 2, while deferring coverage of the remaining solutions to Section 5. Section 3 describes how to assign current-time timestamps to transactions, detailing first a solution that delays this assignment as long as possible. This solution is then refined, enabling the initial assignment of ranges of timestamps, as this further improves performance. The solution uses read timestamps in addition to the write timestamps that are required for timestamping of the data. Section 4 offers techniques that aim at offering better performance in the management of these read and write timestamps. Finally, Section 6 summarizes and points to directions for future research.

2 Prior Timestamping Approaches

Let us begin by describing prior timestamping techniques. Our approach, while new both in the specific problem it solves and in its major elements, borrows selected elements of these prior techniques. It is useful to describe these early, both to help in describing how these elements attack the problem and to contrast these approaches with ours.

2.1 Commit Time Choice of Timestamp

If we are willing (or able) to delay the choice of timestamp until transaction commit then it is possible to very simply choose a timestamp. We simply choose a timestamp that reflects the order in which transactions have committed. That is, we issue the transaction a timestamp that reflects its time of commit. This requires that we use something other than timestamp ordering for our concurrency control technique, such as two-phase locking.

In [8], timestamps that agreed with transaction serialization order were used to optimize two-phase commit (2PC). A single variable *LAST* was maintained by a database, representing the time that the last transaction was committed. Each subsequent transaction that attempts to commit was given a timestamp greater than *LAST*, and *LAST* was updated to that later timestamp.

The problem we have with late timestamp choice is that it does not permit us to be responsive to a request, by a statement in a transaction, for the "current time." Such a request means that we cannot simply choose a time at commit, but need to make our choice when the request occurs. We can adapt our *LAST* method to the case where a transaction's timestamp may be chosen earlier as follows. As each subsequent transaction with $t > \mathit{LAST}$ commits, we set *LAST* to the new (larger) t value. We abort transactions with timestamps $t < \mathit{LAST}$. This enforces that timestamp order agree with all conflict orders between transactions, including read-write conflicts. Unfortunately, it also has the effect of aborting a large number of transactions.

2.2 Timestamp Concurrency Control

Timestamp order concurrency control (TO) enforces that transactions commit in timestamp order—when the commit of a transaction will violate this order, the transaction is aborted. In our first attempt above, we bounded the timestamps of all prior transactions in a single variable *LAST*, a worst case value. With TO, we are much more precise about the timestamps of prior conflicting transactions, and we can hence provide larger acceptable ranges of values for most transactions' timestamps.

More specifically, TO, which has existed for about 20 years [1], chooses the timestamp t for a transaction at its start (or at least by the time of its first data access) and assigns this timestamp to data items when they are read as well as when they are written. Transactions can access data in conflicting ways, and a solution that keeps transaction timestamps consistent with a transaction serialization order needs to handle three forms of conflict: write-write, write-read, and read-write. TO associates a write timestamp ($d.\mathrm{TT}^{\vdash}$) and a read timestamp ($d.\mathrm{T}^{\mathrm{R}}$) with each data item d, hence trying to minimize aborts resulting from transaction ordering conflicts by maintaining precise timestamp constraints for each transaction.

In transaction-time databases, the value $d.\mathrm{TT}^{\vdash}$ is stored persistently with the data, to capture previous states and thus support timeslice queries. This provides the write timestamps that allow us to handle write-write and write-read conflicts. But to handle read-write conflicts requires that we also maintain read timestamps $d.\mathrm{T}^{\mathrm{R}}$. Hence, whenever d is read by transaction X with timestamp t_X that is larger than d's current $d.\mathrm{T}^{\mathrm{R}}$ (and thus later than the times of its earlier readers), $d.\mathrm{T}^{\mathrm{R}}$ is set to t_X. $d.\mathrm{T}^{\mathrm{R}}$ does not need to be a persistent part of the database, as queries do not ask about times when data is read. So TO methods can exploit a volatile data structure that captures $d.\mathrm{T}^{\mathrm{R}}$ for each d.

A transaction compares its timestamp t_X to the data item d's read timestamp $d.\mathrm{T}^{\mathrm{R}}$ in the volatile structure and its write timestamp $d.\mathrm{TT}^{\vdash}$, which is stable in a transaction-time database, to determine timestamp consistency. We are only concerned with current data items, i.e., items with $\mathrm{TT}^{\dashv}$ value equal to now, because these are the only items that can be written.

- When transaction X attempts to read data item d, we require that $t_X > d.\mathrm{TT}^{\vdash}$. Further, if $t_X > d.\mathrm{T}^{\mathrm{R}}$, then $d.\mathrm{T}^{\mathrm{R}}$ is set to t_X, to permit us to validate updater timestamps (see the next item). If $t_X < d.\mathrm{TT}^{\vdash}$, a later transaction wrote item d, so the transaction came too late and aborts.

- When transaction X attempts to write data item d, we require that $t_X > d.\mathrm{T}^{\mathrm{R}}$ and $t_X > d.\mathrm{TT}^{\vdash}$ as well. Otherwise, transaction X aborts. This enforces both write order and that a previous reader of d read the correct version.

TO presents two problems in our context.

1. Each transaction X needs to have a timestamp t_X when it starts, so that it can use t_X in the timestamp consistency testing above. However, timestamps should be chosen as late as possible in the transaction's execution, ideally when the transaction is about to commit, so as to minimize the chance of abort. At that point, t_X can be chosen to be consistent with the order in which the transaction is committing. By forcing the choice of t_X at transaction start, one significantly increases the chances that the timestamp consistency checking will fail, resulting in an abort.

2. While $d.\mathrm{T}^{\mathrm{R}}$ needs not be stable, the number of data items is potentially enormous. The access structure for the $d.\mathrm{T}^{\mathrm{R}}$'s needs to provide efficient access for all current (or at least a large number of) data items d, and needs to be a dynamic structure that can grow in size as more data items are included in it. At some point, its growth needs to be limited so that it can be reasonably maintained in volatile memory. TO methods have exploited "garbage collection" to prune this structure. The garbage collection technique in [1] deletes timestamps that are older than some δ. That is, any read timestamp $d.\mathrm{T}^{\mathrm{R}} < (t_{current} - \delta)$ is deleted. If a transaction with a timestamp less than $(t_{current} - \delta)$ is still active and references a data item with no stored timestamp, it is aborted, as all data items without an explicit read timestamp are implicitly given a read timestamp equal to $(t_{current} - \delta)$. This risks additional aborts.

The solution presented in the remainder of the paper avoids the first problem of selection a transaction's timestamp at transaction start, as well as the second problem of maintaining a potentially enormous structure for the read timestamps of transactions.

3 Deferring Timestamp Choice

This section describes how it is possible to delay the choice of a transaction's timestamp, until the moment when the timestamp is needed by a statement in the transaction or until the transaction commits. We detail the actions that must be taken before a timestamp is assigned, the procedure for assigning a timestamp, the actions that must be taken after a timestamp is assigned, and the procedure for commit processing. Section 3.2 improves on this machinery, by allowing the use of initially imprecise timestamps.

3.1 Late Timestamp Assignment

Each data item d in a transaction-time database has a write timestamp $d.\mathrm{TT}^{\vdash}$, which is set to the timestamp of the transaction that created it. We also assume that a read timestamp $d.\mathrm{T}^{\mathrm{R}}$ is associated in some way with each current data item via a volatile data structure, which we define and consider in detail in Section 4.

During the execution of a transaction, two-phase locking (2PL) [1, 5] blocks transactions from reading data that is being written by an uncommitted transaction, and from writing data being read or written by an uncommitted transaction. Thus, timestamps need play no role in protecting data of an active transaction. This is essential to our strategy of delaying the assignment of timestamps. Only after commit is it essential that timestamps be associated with data, to ensure that subsequent transactions have timestamps that are consistent with the ordering resulting from the write-write, read-write, and write-read conflicts.

We need to describe what happens before a transaction has a timestamp, how a timestamp is assigned, what happens after the transaction has a timestamp, and what we need to do at commit time to ensure that the timestamps of subsequent transactions are appropriate. Our initial description will assume that, when a transaction requests the current time, it receives back a full precision time that will be used as its timestamp. We subsequently discuss how to relax this requirement.

During the execution of a transaction, we need to retain information to use when selecting its timestamp, and to provide information that governs the selection of the timestamps for other transactions. Data accesses define a lower bound t_l on the value that can be chosen as the transaction's timestamp. We initialize t_l to t_s, the execution start time for the transaction, so that the chosen timestamp will not be before the start of the transaction. We also remember the set of items V_R read by the transaction, the set V_I of new items inserted by the transaction, and the set V_D of items deleted by the transaction. An update is treated as a delete followed by an insert in the same transaction. These sets are initialized to the empty set, $\emptyset$.

3.1.1 Before Timestamp Assignment

Before a timestamp has been assigned, a transaction is never aborted due to timestamping constraints because these are never violated. A timestamp can always be chosen later that satisfies the necessary constraints. So, if we are not forced to choose a timestamp, we delay as long as possible in providing a timestamp for the transaction. This delay reduces the chances that the transaction will be aborted because of timestamp consistency requirements. However, even before timestamp assignment, we need to update information that determines the constraints that the eventual timestamp must satisfy. This updating occurs when our transaction X attempts to read or write a data item.

Read(d): If d is locked because of an active transaction write, the transaction blocks. Once we can access d, we need to ensure that our transaction will not be given a timestamp $t_X \leq d.\mathrm{TT}^{\vdash}$. Thus, if $d.\mathrm{TT}^{\vdash} > t_l$, then $t_l \leftarrow d.\mathrm{TT}^{\vdash}$. We set $V_R \leftarrow V_R \cup \{d\}$ to remember that we have read d, for commit-time processing.

Write(d): If d is locked because an active transaction is reading or writing d, then the transaction blocks. Once we can access d, we need to ensure that our transaction will not be given a timestamp $t_X \leq \max(d.\mathrm{T}^{\mathrm{R}}, d.\mathrm{TT}^{\vdash})$. Thus, if $\max(d.\mathrm{T}^{\mathrm{R}}, d.\mathrm{TT}^{\vdash}) > t_l$ then $t_l \leftarrow \max(d.\mathrm{T}^{\mathrm{R}}, d.\mathrm{TT}^{\vdash})$. We set $V_I \leftarrow V_I \cup \{d\}$ when the write is an insert to remember that we have inserted d, and we set $V_D \leftarrow V_D \cup \{d\}$ when the write is an delete to remember that we have deleted d, again for commit-time processing.

3.1.2 Timestamp Assignment

When we choose a timestamp t_X for our transaction X, we choose it so that all accesses to data within the transaction satisfy the following constraints. This is always possible.

Reads: $t_X > d.\mathrm{TT}^{\vdash}$ for all $d \in V_R$. This constraint enforces write-read conflicts. Note that, because reads are commutative, we do not require that $t_X > d.\mathrm{T}^{\mathrm{R}}$.

Writes: $t_X > d.\mathrm{T}^{\mathrm{R}}$ and $t_X > d.\mathrm{TT}^{\vdash}$ for all $d \in V_I \cup V_D$. These constraints enforce read-write and write-write conflicts, respectively.

This information is captured in the variable t_l, which serves as a lower bound for the value of the timestamp t_X that we can assign to the transaction. Since we have not (yet) provided an upper bound for t_X, we can at this point always assign an acceptable timestamp, i.e., one that satisfies our timestamp constraints.

A transaction may be assigned a timestamp in the midst of its data accesses. So a transaction can change from one without a timestamp to one with a timestamp at any time. This is important in our effort to delay the choice of timestamp. Only when a request for the current time, e.g., CURRENT_TIME in SQL [10], is made are we compelled to assign a timestamp prior to commit. Otherwise, we can wait until transaction commit, at that time assigning a timestamp that agrees with transaction conflict order.

Monotonically increasing timestamps in which the timestamp assigned is greater than all previously issued timestamps will provide us with a t_X that satisfies all prior timestamp constraints. However, we achieve tighter bounds by remembering the largest $d.\mathrm{TT}^{\vdash}$ for any d that has been accessed, and the largest $d.\mathrm{T}^{\mathrm{R}}$ for any d that has been updated, which is what we do by maintaining variable t_l. Then we set the timestamp $t_X > t_l$.

It is especially useful to select a minimum t_X when timestamp assignment immediately precedes commit as there will be no further opportunity for this transaction to violate timestamp constraints. An earlier time will improve the chances that other transactions will satisfy their timestamp constraints because earlier timestamps make it easier for a later transaction Y to find a timestamp t_Y for itself that is greater.

3.1.3 After Timestamp Assignment

After the timestamp for the transaction is fixed, we now run the risk that timestamp constraints will be violated. As before, let the timestamp of a transaction X be t_X. We now describe how this transaction proceeds as it reads and writes data items. Basically, we abort the transaction if a timestamp constraint is violated.

Read(d): If d is locked because of an active transaction write, the transaction blocks. Once our transaction is able to access d by placing a read (share mode) lock on d, we then perform our timestamp check. If $t_X < d.\mathrm{TT}^{\vdash}$ then we abort the transaction. Otherwise, we proceed as usual. We set $V_R \leftarrow V_R \cup \{d\}$ to remember that we have read d, for commit-time processing.

Write(d): If d is locked because an active transaction is reading or writing d, then the transaction blocks. When our transaction is able to access d by placing a write (exclusive mode) lock on d, we then perform our timestamp check. If $t_X < d.\mathrm{T}^{\mathrm{R}}$ or $t_X < d.\mathrm{TT}^{\vdash}$ then we abort the transaction. Otherwise, we proceed as usual. For inserts, we set $V_I \leftarrow V_I \cup \{d\}$ to remember that we have inserted d, and for deletes, we set $V_D \leftarrow V_D \cup \{d\}$ to remember that we have deleted d, once again for commit-time processing.

3.1.4 Commit Processing

At commit, we need to ensure that the database and our auxiliary data properly reflect that transaction X with timestamp t_X has committed. This involves posting timestamp information with the data items that have been read and written by the transaction, so as to ensure that subsequent transactions can be assigned appropriate timestamps. The following is required.

Reads: Set $d.\mathrm{T}^{\mathrm{R}} = t_X$ for all data items $d \in V_R$ (those we have read) for which $d.\mathrm{T}^{\mathrm{R}} < t_X$. Note that we are allowed to make $d.\mathrm{T}^{\mathrm{R}}$ for other d's greater than they need to be. This can be a "conservative" action, as $d.\mathrm{T}^{\mathrm{R}}$ does not play a permanent role in a transaction-time database, but is only used to enforce the read-write ordering. If the transaction aborts, nothing needs be done about data items that were read.

Writes: These are used for inserts and deletes. For each $d \in V_D$, those items we have deleted from the current state, we set $d.\mathrm{TT}^{\dashv} = t_X$. For each $d \in V_I$, those items that we inserted into the current state, we set $d.\mathrm{TT}^{\vdash} = t_X$ (and $d.\mathrm{TT}^{\dashv} = now$). Table 1 describes the use of writes. If the transaction aborts, we do not revisit d, at least in some scenarios. We will describe this briefly in the next section.

3.2 Incremental Timestamp Refinement

As with delaying timestamp choice, keeping the timestamp as imprecise as possible reduces the likelihood of transaction abort. Up until now, we have merely put a lower bound

user-level operation	low-level operation(s)
delete d	$w(d, old, [t, t_X))$
insert d	$w(d, new, [t_X, now))$
update d	$w(d, old, [t, t_X))$ $w(d, new, [t_X, now))$

Table 1: Writes by a Transaction X with Timestamp t_X

t_l on the timestamp t_X that we choose for transaction X. While there was, perhaps, an implicit assumption that we chose the precise timestamp t_X when the current time was requested, or at commit time, we have not discussed a specific strategy for selecting a transaction timestamp.

What we describe here is how, instead of fully specifying a timestamp for a transaction when the current time is requested, we instead use the request to provide an upper bound on the value of $t_{current}$ for transaction X, and hence on its timestamp t_X. Only when a fully precise time is specified is $t_{current}$ fully specified, and hence t_X fully determined. This further exploits late timestamp choice, extending it to permit this choice to be refined incrementally during transaction execution. Importantly, the choice usually need not be completely defined until the transaction commits. At commit, we must post the timestamp to data items that have been read and written by the transaction, and so require a precise timestamp at that point.

Working with a timestamp range that has upper bound t_h as well as a lower bound captures this imprecision. The initial value for t_h is ∞.

3.2.1 Impact of Time Requests

When a transaction in an SQL-compliant database requests the current time, it is possible to request this time with a designated precision. For example, if the transaction asks for CURRENT_DATE, only the day is provided. If CURRENT_TIME or CURRENT_TIMESTAMP is requested, one can specify a precision. Whatever precision is requested, the result is to provide candidate upper and lower bounds.

The result of a current-time request is whatever the system clock says it is, truncated to the specified precision. The lower bound is that time extended by ‘0’b’s to the maximum time precision, provided it is greater than the previous lower bound; the upper bound is that time extended by ‘1’b’s to the maximum time precision, provided it is less than the previous upper bound.

For example, if CURRENT_DATE is requested then the timestamp upper bound becomes the last time instant (at maximum precision) for the day that is returned. The lower bound constraint provided by this time request is the first instant of the day. When CURRENT_TIME or CURRENT_TIMESTAMP are requested at less than maximum precision, similar considerations apply.

We now relate timestamp ranges to our previous protocol. Instead of assigning a fully precise timestamp during timestamp assignment, we assign a timestamp range, with upper and lower bounds as just described, which we denote by $\langle t_l, t_h \rangle$. After *timestamp range assignment*, a transaction may continue to access data, via reading and writing. As previously, we abort transactions that cannot satisfy the timestamp constraints, now expressed as bounds on the timestamp. A transaction aborts whenever $t_l \geq t_h$.

3.2.2 Committing with Timestamp Ranges

At commit time, it is advantageous to choose transaction X’s timestamp, if not yet specified with full precision, to be $t_X = t_l^+$, one unit larger at the finest precision available than the lower bound of the timestamp range. This choice is acceptable for transaction X, and makes it easier for other transactions to find an acceptable timestamp as it minimizes the value of t_X. Hence it permits subsequent transactions that read or write this data to preserve somewhat larger timestamp ranges, which increases the probability that they will escape timestamp-induced aborts.

When X is a distributed transaction, it is possible to use a timestamping commit protocol to determine a transaction timestamp for all transaction participants [8]. Each participant’s “prepared” vote includes a timestamp range within which the participant guarantees that it can commit. The coordinator intersects all voted ranges, and then chooses a time t_X within that range as the timestamp for the transaction. Normally t_X will again be t_l^+.

4 Timestamping Data

The goal that led us to assigning timestamps to transactions is to provide transaction-time database functionality, meaning to retain all previously current database states, making them available for queries such as “what was the balance in John’s checking account on June 30, 1999?” To provide this functionality, we must

- associate a pair of transaction timestamps with each data item in the database, and
- choose timestamps for transactions so that the timestamps reflect a serialization order for transactions.

Specifically, timestamps $d.\mathrm{TT}^{\vdash}$ and $d.\mathrm{TT}^{\dashv}$ must be associated with all data in a transaction-time database. In addition, we need $d.\mathrm{TT}^{\vdash}$ and $d.\mathrm{T}^{\mathrm{R}}$ to ensure that timestamp order agrees with serialization order. It follows that read timestamps $d.\mathrm{T}^{\mathrm{R}}$ are only needed for the timestamp assignment process, and we can dispense with them once their role in that process has been completed.

Providing and managing read timestamps $d.\mathrm{T}^{\mathrm{R}}$ is the more challenging issue, so we first describe a new approach for this that has some significant advantages. While there are issues associated with the timestamps $d.\mathrm{TT}^{\vdash}$ and $d.\mathrm{TT}^{\dashv}$ of data items, which we address at the end of this section, we can exploit existing techniques for this, though we do suggest an improvement on the method that we prefer.

4.1 Read Timestamps

We have only described abstractly that the system maintains a $d.\mathrm{T}^{\mathrm{R}}$ value for each data item d that is read by a transaction. Clearly we need to deal with this concretely. Ideally, we prefer a technique for managing these values that is simple to implement, has high performance, and minimizes aborts. Thus, we would like an approach that is more flexible than the *LAST* technique and lower in overhead than the prior TO approach. We propose such an approach, then consider the impact of system crashes.

4.1.1 The Read Timestamp Table (RTT) Approach

We must (conservatively) enforce that timestamp order is consistent with read-write conflict order. This leads us to suggest that the T^{R} values be provided via a hash table that we call the Read Timestamp Table, or *RTT* for short, where each entry determines a T^{R} value for a set of data items D; i.e., we map a set of d's (via, e.g., a hash function) to an identifier I_D for the set D, i.e., $hash(d) = I_D$. I_D then is used to access *RTT* and determine the common T^{R} value for all members of D, i.e., *RTT*$(hash(d))$. The *RTT*'s size can be varied depending upon the desired trade-off between storage overhead and abort rate. The larger the table, the smaller is each set D that is coalesced and managed with the same T^{R} value.

Each data item d that a transaction X writes requires that we check that the read-write conflict order between X and earlier read transactions agrees with their timestamp order. We can only do this when we know the timestamp t_X for X. Hence, writes require that we check the *RTT*. Each data item that X reads requires that we update the *RTT* so as to be able to subsequently ensure that the read-write conflict order between X and subsequent write transactions agrees with the timestamp order.

We cannot check or update *RTT*, however, until X has been given a timestamp. Fortunately, while X is executing, locking ensures serializability of transactions, and transaction atomicity ensures that we, as late as commit time, can enforce timestamp order, by transaction abort if necessary. So let us first describe how all *RTT* checking and updating can be done at commit time.

To perform the appropriate read timestamp checking at commit time, we need only remember the hash values $i = hash(d)$ for all d that were written by the transaction. With an *RTT* that consists of a few hundred entries, we can remember this set via a bit vector V_W denoting the variables written, which reflects the hash values for all items in $V_D \cup V_I$. An *RTT* with 512 entries would then require a V_W of 64 bytes for each transaction. Each entry *RTT*$[i]$ in which $V_W[i]$ =‘1’b is accessed at commit time and compared with the transaction X's timestamp t_X. If any of the *RTT*$[i] \geq t_X$, we abort the transaction.

To perform the appropriate read timestamp updating at commit time, we likewise need only remember the hash values $i = hash(d)$ for all d that were read by the transaction. This is a second bit vector V_R of, e.g., 64 bytes, yielding a total bit vector space overhead of 128 bytes. So the overhead need not be large. If $t_X >$ *RTT*$[i]$ when $V_R[i]$ =‘1’b, then *RTT*$[i] \leftarrow t_X$.

It is possible to check and update entries in *RTT* as soon as a transaction has a timestamp. Indeed, when checking *RTT*, it is possible to detect the need to abort a transaction as soon as there is an upper bound t_h such that $t_h < \mathrm{T}^{\mathrm{R}}$. However, the better question to ask is “when is it desirable to check or update *RTT*?” Our answers follow.

updating ***RTT*****:** It is never desirable to update *RTT* earlier than commit time. Read locks prevent subsequent writers from writing any read data item d, so there is no possibility of a timestamp order violation involving d until after a transaction commits. Indeed, early update of *RTT*$[hash(d)]$ will increase the number of false timestamp order violations because the timestamp associated with *RTT*$[hash(d)]$ will associate the later timestamp (of d) with all data items in D.

checking ***RTT*****:** It is always desirable to check *RTT* as soon as a transaction X has a fully specified timestamp t_X. Suppose that X wrote data item d. X's write lock prevents any subsequent (later) reader from reading d. So a precise $d.\mathrm{T}^{\mathrm{R}}$ cannot increase in value for the duration of the transaction. However, the value of *RTT*$[hash(d)]$ can be updated because some other transaction reads a d' with $hash(d') = hash(d)$, and this might lead to an unnecessary abort of X. By checking *RTT*$[hash(d)]$ as early as possible, we reduce the window in which this kind of update can occur. So, at timestamp assignment time, it is desirable to use V_W to check the entries of *RTT* immediately. In addition, when there is a subsequent write of a data item d', we check *RTT*$[hash(d')]$ immediately, without exploiting our bit vector. This early check also identifies transactions that need to be aborted as early as possible, meaning that there is less wasted work done by the doomed transaction.

4.1.2 Impact of System Crashes

We need to understand the impact that a system crash has on our method. Since the *RTT* is used in an essential way only to help choose the timestamps of transactions, the *RTT* does not need to be stable so long as we can ensure that we can identify the latest possible timestamp *LAST* (recall *LAST* from Section 2) used by any committed transaction prior to the crash. This information must be available in the recovery log, as we must be able to timestamp versions of data written by every committed transaction, i.e., to write for each written d also its $d.\mathrm{TT}^{\vdash}$ (and perhaps its $d.\mathrm{TT}^{\dashv}$).

Transactions that were active at the time of the crash are aborted, so their activities have no effect. We initialize all entries of *RTT* with *LAST*. While this is conservative, in that the pre-crash *RTT* entries might have had earlier timestamps, no additional local transactions will be aborted by this process, as all post-crash local transactions will have timestamps greater than *LAST*.

We need *LAST* (rather than using, e.g., *ZERO*) because it is possible that some non-local distributed transactions have timestamps earlier than *LAST*. Using *LAST* guarantees that we prevent read-write order violations, whereas

ZERO does not guarantee that. These distributed transactions may be aborted because read timestamps are now later than they might otherwise have been. Thus, a local cohort of a distributed transaction would surely have *LAST* as a lower bound on its commit time. A distributed transaction X might, at another of its cohorts, be forced to choose a $t_h <$ *LAST*, e.g. because of a CURRENT_TIME request by the cohort, thus requiring that the transaction be aborted. But this is a low probability occurrence, in any event.

The bottom line is that there is little point in making the *RTT* stable. Correct behavior is always (conservatively) assured, and the probability of extra aborts is low.

4.2 Write Timestamps

For completeness, we describe here how to handle write timestamps, summarizing previous published solutions that we believe are effective. As with read timestamps, we consider the impact of system crashes.

4.2.1 Second Access for Timestamping

Timestamps $d.\mathrm{TT}^{\vdash}$ are stored with each data item d, denoting the time at which each is inserted. Transaction-time databases store in addition an end time for their data items, as described in Section 1. Thus, with each data item d is also stored $d.\mathrm{TT}^{\dashv}$ denoting the time at which it is deleted. A write transaction has to post both new data values and these timestamps. Since we may not have the transaction's timestamp when the write occurs, we need to record these writes in some other way, completing the writes when we have sufficient information.

One way to deal with this is to create an *intentions list* of inserted or deleted data items that records both the identities and values of the items each transaction X writes. The writes remembered in the intentions list are then posted when sufficient information is later available. Such an intentions list has two disadvantages, however.

1. Data values can be very large, and so storing intentions lists for all uncommitted data can be costly in main memory consumption.

2. A transaction is expected to see its own updates. With the intentions list approach, every data access needs to look at the intentions list to determine whether the transaction has previously updated a data item.

We avoid intentions lists by posting an update in the appropriate database page at the time of the write. The 2PL protocol protects these uncommitted and yet-to-be-timestamped writes. Since a transaction X does not necessarily know its timestamp early in its execution, these data items are initially tagged with X's *transaction id* when the write occurs. We re-visit these data items later, when we know X's timestamp t_X.

We are not guaranteed to know X's timestamp until it commits. For a distributed transaction, it may be impossible to choose a timestamp prior to execution of the commit protocol when all transaction cohorts "vote" on commit and perhaps on transaction time [8]. Hence, we re-visit the data items written by committing transaction X after it commits and we are guaranteed to know its timestamp value t_X. Placing t_X in X's commit record ensures that the connection between transaction id and timestamp is stable. Then data items inserted (or deleted) are re-visited, and transaction id is replaced by t_X. In the normal course of system operation, pages containing these records "find their way" to the disk and become stable in that way.

We strongly endorse this "traditional" approach. However, note that posting timestamps to written data items after transaction commit leads to a double access to all written data items, once to post the write and, after commit, to post the transaction timestamp that overwrites the transaction id.

Finally note that the description here has been phrased mostly in terms of a data item d having a timestamp that marks the time of its writing transaction. The time of the writing transaction is used as the $\mathrm{TT}^{\vdash}$ value for a new data item d' when an insertion occurs (with the end time $\mathrm{TT}^{\dashv}$ having the variable now as its value). The timestamp for the writing transaction is used as the (new) $\mathrm{TT}^{\dashv}$ value when an existing data item is being deleted. An update is accomplished by two writes, one to delete the existing data item d and one to insert the new data item d'. We have been maintaining the set of inserted items V_I and deleted items V_D to enable this commit time processing.

4.2.2 Impact of System Crashes

Unlike the situation with read timestamps, write timestamps for data need to be permanently assigned to the data written by every committed transaction. System crashes can interfere with the process of associating timestamps with data. We need to ensure that the timestamps that are needed in order to support the desired transaction-time database functionality after a crash indeed survive the crash. And we need to identify those timestamps that are no longer needed, and justify that they are not. Storing a transaction's timestamp in its commit record is a start at making its timestamp stable, but only a start.

Here we summarize two previous approaches [12, 13] for how to complete the timestamping of transactions after commit. Both provide a persistent table that maps committed transaction id's to their corresponding timestamps, which we call the timestamp table or *TST*. The approaches differ in how they manage the *TST* table subsequently.

1. In [13], *TST* contains the timestamps for all transactions. When an unstamped data item is read, *TST* is consulted and the transaction id is replaced with the timestamp found in the *TST*. The difficulty with this approach is that the *TST* can get very large, and accessing it can add to disk I/O.

2. In [12], only the transaction entries needed for unstamped data are maintained in the *TST*. Entries for which timestamping is complete are garbage collected. There is both an execution cost and a system

complexity cost to *TST* garbage collection, as the system needs to maintain information about unstamped data to know what *TST* entries to keep or purge.

On balance, it seems better to us to do the garbage collection and prune the *TST*.

Garbage Collecting *TST* Entries

In [12], the identity of each unstamped data item d is maintained in a list associated with the timestamp that is needed by d. As data items are timestamped, they are removed from the list. The subtlety, as explained in [12], lies in ensuring that this bookkeeping information is itself persistent. And the advantage of retaining this list of data items is that we can choose, via referencing the data items, to complete the timestamping and remove the associated timestamp entry from the *TST* at any time.

However, if we are willing to accept a slightly larger *TST*, where we cannot simply complete the timestamping at our discretion, we can replace the list of items with a count of the objects that remain to be timestamped. When a page is written to disk, we create a log record that describes the write and its impact on timestamp counts. This log record contains, for each timestamp, the number of previously unstamped data items in the page that were given that timestamp. Thus, we can (almost always) persistently maintain counts of the unstamped data for each timestamp. If a crash occurs between the write of a page and the write of the log record describing the write, the only result is that some timestamp entries in the *TST* are not garbage collected, which is probably not very important.

We can make persistent reference counting precise by writing to the log our intention to write pages in a "system transaction" start log record, then writing the pages, and finally committing this "system transaction" after all listed pages are stable on the disk. We include with the list of pages being written the changes in reference counts for unstamped data in the start log record, and would redo the "transaction" were the system to crash and there were no commit record in the stable log.

5 Related Work

The predominant assumption in past work on temporally enhanced data models and query languages as well as implementation techniques for temporal data management [7, 11] has been that there is no support for user-specified transactions, making individual modification statements the units of work. In perhaps the most relevant contribution based on this assumption, Clifford et al. [2] offer a semantic foundation for associating the variable *now*, denoting the current time, with data items. Our proposal follows this foundation, but assumes the presence of transactions.

Transaction support in temporal databases has only been addressed in a handful of works. We review these next, also describing how this paper's contribution builds on and extends these.

The Postgres DBMS [13, 14] supports transaction time by timestamping after commit. Specifically, Postgres uses eight extra attributes in each relation in order to associate with each database modification both the id and commit time of the transaction that effected the modification. Initially, only the transaction id is associated with a modification. The transaction-time values are left unassigned when a tuple is initially stored in the database [13]. When a transaction commits, its commit time is stored in a special `Time` table. This enables the subsequent revisitation of tuples in order to apply the permanent timestamps. This revisitation is done lazily or not at all [13]. When the transaction time of a tuple is needed, but is not stored in the tuple, the `Time` table is consulted. There is no discussion of the use of temporary values of the timestamp attributes in Postgres.

Salzberg [12] demonstrated that to achieve a transaction-consistent view of previous database states, it is necessary to use the same timestamp for all modifications within a transaction. Further, this 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. Again, the use of current time prior to commit is not considered.

The techniques proposed in our paper follow the principles put forward by Salzberg and use principles similar to those used in Postgres, but extends these to solve the more general problem encountered in practice. In particular, we deal with the fact that user requests can force an early choice of timestamps.

Finger and McBrien [4] studied timestamping, including the use of the valid-time variable *now*. Like Salzberg, 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 as the value for *now*. They showed that this choice may lead to *now* appearing to be moving backwards in time and 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 differs quite substantially from our solution where we use the commit time. Further, it has been shown that ignoring the phenomenon of *now* moving backwards may lead to violation of the isolation principle of transactions and that transaction executions cannot be serializable in the order of their start times, if concurrency is to be allowed [16].

In recent related work, Torp et al. [16] consider the timestamping of data items assuming a layered architecture, where support for a temporal query language is provided, essentially, by an application that uses the SQL interface of a conventional DBMS. With no access to the internals of the DBMS, the possible solutions are more restricted. They use the commit time of a transaction as the current-time value given to all requests for the current time from statements in the transaction. As in our paper, and unlike Salzberg and Stonebraker, they permit statements in a transaction to reference current time. Since the commit time is unknown until the transaction has exhausted all its statements and is ready to commit, a single, temporary current-time value is used, namely the system time when the current time is first needed by a statement in the transaction. Techniques that generalize the revisitation approach outlined for Postgres are supported.

While performance experiments demonstrate that this timestamping approach has good performance, the restrictions imposed by the assumed architecture renders the approach only approximately correct. In contrast, our paper, by augmenting the two-phase locking mechanism with read timestamps, ensures correctness.

Finally, the values $t_{current}$ and now used, but specified only abstractly in our paper may be represented and manipulated as described in detail in [16].

6 Summary and Research Directions

In practical database applications, it is frequently important to retain a perfect record of past database states. For example, this need mirrors the practice in finance where accountants correct errors, not by using an eraser, but by posting compensating transactions to the books. In medical applications, documenting the basis on which decisions were made by such a record may guard against wrongful malpractice lawsuits.

By building on past work in concurrency control, recovery, and temporal databases, we provide the first full solution to the problem of correctly supporting transaction timestamping of databases in a realistic setting, where user-specified transactions are allowed and concurrency control and recovery are accomplished using two-phase locking and logs, respectively. Particular care has been given to obtain a solution that does not significantly decrease the level of concurrency and number of aborts of a two-phase locking system, that only introduces modest bookkeeping, that ensures that recovery is accounted for, and that is consistent with the SQL standard.

Specifically, our solution to transaction timestamping generalizes the approach of choosing the transaction timestamp at commit time, enabling a transaction to choose its timestamp at any time after transaction start, which is required because SQL transactions may request the “current time.” It permits us to delay a transaction’s choice of a timestamp as long as possible, reducing the risk that transactions may need to be aborted in order to make the choices of timestamps consistent with a serialization order. To further reduce the likelihood of aborts, imprecise timestamps may be used initially. Next, to handle read-write conflicts, in addition to write-read and write-write conflicts, our solution uses a simple and flexible auxiliary structure with read timestamps.

As future research, it is of interest to ensure independence of the granularity of the time domain, which we believe may be achieved by enabling the solution to allow non-decreasing timestamps to be assigned to consecutive transactions instead of strictly increasing timestamps.

Acknowledgments

The authors would like to thank Richard Snodgrass for his participation in the research that led to this paper, including particularly his role in initiating this work.

Christian S. Jensen was supported in part by the Danish Technical Research Council through grant 9700780 and by a grant from the Nykredit corporation.

References

[1] P. Bernstein,, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, 1987.

[2] J. Clifford, C. E. Dyreson, T. Isakowitz, C. S. Jensen, and R. T. Snodgrass. *On the Semantics of ‘Now’ in Databases*. ACM TODS, 22(2):171–214, June 1997.

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

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

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

[6] C. S. Jensen and C. E. Dyreson (eds). The Consensus Glossary of Temporal Database Concepts. In [3], pp. 367–405, 1998.

[7] C. S. Jensen and and R. T. Snodgrass. Temporal Data Management *IEEE TKDE*, 11(1):36–44, January/February 1999.

[8] D. Lomet. Using Timestamps to Optimize Two Phase Commit. *Proceedings of the PDIS Conference*, January 1993, pp. 48–55.

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

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

[11] G. Özsoyoğlu and R. T. Snodgrass. Temporal and Real-Time Databases: A Survey. *IEEE TKDE*, 7(4):513–532, August 1995.

[12] B. Salzberg. Timestamping After Commit. *Proceedings of the PDIS Conference*, September 1994, pp. 160–167.

[13] M. Stonebraker. The Design of the POSTGRES Storage System. *Proceedings of the VLDB Conference*, September 1987, pp. 289–300.

[14] M. Stonebraker, L. A. Rowe, and M. Hirohama. The Implementation of Postgres. *IEEE TKDE*, 2(1):125–142, March 1990.

[15] A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. T. Snodgrass, (eds) *Temporal Databases*. Benjamin/Cummings, 1993.

[16] K. Torp, C. S. Jensen, and R. T. Snodgrass. Effective Timestamping in Databases. *VLDB Journal*, 8(3–4):267–288, 2000.

[17] C. Vassilakis, N. A. Lorentzos, and P. Georgiadis. Implementation of Transaction and Concurrency Control Support in a Temporal DBMS. *Information Systems*, 23(5):335–350, 1998.

[18] Y. Wu, S. Jajodia, and X. S. Wang. Temporal Database Bibliography Update. In [3], pp. 338–366, 1998.