©2005 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

This material is presented to ensure timely dissemination of scholarly and technical work. Copyright and all rights therein are retained by authors or by other copyright holders. All persons copying this information are expected to adhere to the terms and constraints invoked by each author's copyright. In most cases, these works may not be reposted without the explicit permission of the copyright holder.

# Using the Lock Manager to Choose Timestamps

*David Lomet*
*Microsoft Research*
*Redmond, WA, USA*
lomet@microsoft.com

*Richard T. Snodgrass*
*University of Arizona*
*Tucson, AZ, USA*
rts@cs.arizona.edu

*Christian S. Jensen*
*Aalborg University*
*Aalborg, Denmark*
csj@cs.auc.dk

## Abstract

*Our goal is to support transaction-time functionality that enables the coexistence of ordinary, non-temporal tables with transaction-time tables. In such a system, each transaction updating a transaction-time or snapshot table must include a timestamp for its updated data that correctly reflects the serialization order of the transactions, including transactions on ordinary tables. A serious issue is coping with SQL CURRENT_TIME functions, which should return a time consistent with a transaction's timestamp and serialization order. Prior timestamping techniques cannot support such functions with this desired semantics. We show how to compatibly extend conventional database functionality for transaction-time support by exploiting the database system lock manager and by utilizing a spectrum of optimizations.*

## 1. Introduction

For applications in, e.g., financial and medical domains, accountability and trace-ability are serious concerns. As a reflection of this, it is standard in accounting to post a compensating transaction, rather than performing an in-place update, when an error is discovered. This way, all past states of the accounting records can be reproduced. Transaction-time databases [8, 15] aim to support this type of application. Such a database retains all prior states as well as its current state. It offers a transaction-consistent view of these states, meaning that exactly the states of the database as of any past time are reproducible by means of "as of" queries that take a past time as parameter. It also supports queries that return the sequence of states of some record over some time interval.

Given the enormous investments in existing database applications, it is highly desirable to gradually adopt new transaction-time support [2]. This implies that existing and new transaction-time applications should coexist harmoniously. Thus transaction-time support should be introduced into a DBMS without impacting pre-existing applications. The extended DBMS should not change the semantics of non-temporal queries and updates and existing applications should not experience degraded performance.

We propose to support transaction time functionality at the granularity of a relational table. Those tables for which transaction-time support is needed are specified as transaction-time tables when they are created; conventional and transaction-time tables may coexist. When a transaction-time table is modified, the DBMS timestamps its data. As with concurrency control mechanisms generally, the implementation of transaction-time support may use aborts to ensure correctness. At worst, the user perceives aborts as suboptimal performance.

Support for transaction time is delegated entirely to the DBMS. A transaction-time database maintains the time at which a data item, e.g., relational record, is updated. This is called *timestamping*. Each data item $d$ has two timestamps: that of the transaction whose modification produced $d$, denoted $d.\mathrm{TT}^{\vdash}$ and called the *start time*, and of the transaction whose modification supplanted it, denoted $d.\mathrm{TT}^{\dashv}$ and called the *stop time*. Item $d$ is in the state of the database as of time $t$ when $d.\mathrm{TT}^{\vdash} \leq t < d.\mathrm{TT}^{\dashv}$. An insert statement creates a data item with start time that is the timestamp of the inserting transaction and a stop time of "until-changed," which logically denotes the changing current time [6, 19]; a delete statement will change the stop time from until-changed to the timestamp of the deleting transaction; and an update statement is like a delete followed by an insert.

The choice of the time used in these modification statements is subject to two semantic constraints.

1. The ordering of transaction timestamps must agree with transaction serialization order. If transaction $A$ is earlier in a serialization order than transaction $B$ then $T_A < T_B$, where $T_X$ denotes the timestamp for transaction $X$.
2. The CURRENT_DATE or CURRENT_TIME (SQL nullary functions [13, 17]) within a transaction must return a result consistent with the timestamp of that transaction. While the transaction time has fixed granularity [4, 5], e.g., microseconds, a CURRENT request is at a user-specified granularity, such as DAY or SECOND. Consistency means that the timestamp when CAST to the user-specified granularity is identical to the CURRENT result.

These constraints are challenging to ensure simultaneously, with good performance. Let us briefly consider some of the difficulties.

A transaction is atomic; conceptually all actions of a transaction take place instantaneously. Concurrency control within the DBMS provides this very convenient semantics in the presence of multiple users who simultaneously access and modify the database. Since all actions of a transaction conceptually take place at the same time, this requires the use of the same CURRENT_TIME value for all statements of a transaction. However, the SQL standard allows different statements in the same transaction to use separate CURRENT_TIME values, and which specific values to use are left to the implementation of the database management system.[1]

If a DBMS does not ensure both that a single CURRENT transaction time is used for an entire transaction and that the time chosen is consistent with a valid serialization order, then it is possible that the answer to an "as of" query was never a valid, current state. With SQL-92 [12], a query or modification can reference CURRENT_TIME, and this time can be stored as an attribute in the database or used to query the database, e.g., retrieving the database state current as of this time. This exposes the risk that a query using a transaction's time will not include the results of the transaction whose CURRENT time is used to specify the "as of" time for the query. And if transaction time and serialization order do not agree, the result of such a query may not include all transactions that serialized earlier, and may perhaps include transactions that serialized later.

In this paper, we present techniques to enhance a conventional (non-temporal) DBMS to correctly and efficiently choose transaction timestamps in support of transaction time databases with CURRENT functionality. These techniques extend only the lock manager of the DBMS. Given the complexity of DBMS engines, limiting the impact on DBMS code is important. We believe our incremental approach is essential to enabling adoption of temporal functionality.

## 2. Related Work and Contribution

### 2.1. Earlier Work

Data replication and log analysis tools exist that are capable of extracting data from DBMS logs, thus supporting queries such as a transaction *timeslice* query, a query of a past state of a database [1, 11] that provides an answer based on that past state, as if the past state were the current state. However, tools such as these do not address the core problem of supporting transaction-consistent timestamping. Oracle's recent flashback query facility [20] appears to be better integrated into the DBMS, but it also accesses log data and, again, transaction-consistent timestamping seems not to be part of this facility.

The classic approach to choosing timestamps [14, 18] is to delay the timestamp choice until commit time. Then one can use the transaction's time of commit as its timestamp. In such an approach, termed *late timestamping*, the transaction id is stored in the start or stop time of each data item modified by the transaction. Once the transaction commits, its timestamp is known, and the transaction id within data items is replaced with this time. With strict two phase locking, which we assume, commit order is consistent with transaction serialization order. Hence timestamp order will likewise then be consistent with transaction serialization order, thereby satisfying the first constraint.

[1] The standard fixes the value only within a *statement*, and which fixed value to use is implementation defined. 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." [12, p. 110].

The second constraint is still problematic, because requests for CURRENT TIME can return a value substantially earlier than the commit time, especially for long transactions. A previous approach [9] is summarized in Section 3 and referred to as the *RTT approach*. It satisfied the second constraint, of consistency between the result of CURRENT requests and the transaction's timestamp when only accessing transaction-time data, though with some restrictions which we now point out, and elaborate in Section 3.2.

## 2.2. A New Approach Is Needed

There are two major limitations with the RTT approach, as well as an important aspect was not considered.

**2.2.1. Non-Temporal Data.** Unlike transaction-time data, ordinary data is not timestamped. The problem then is to keep the timestamps that we assign to our temporal data consistent with transaction serialization order when some transactions only access ordinary data, some access temporal data, and some access both kinds of data. The RTT approach did not solve this problem, as it applies only to transactions that access transaction timestamped data. Compatible extension to existing database systems requires a timestamping solution that works when ordinary data may be accessed with transaction-time data in the same transaction. We don't want to restrict applications to accessing only transaction-time tables or only conventional tables.

**2.2.2. Multi-Granularity Locking.** Range queries were not considered in the RTT approach. Support for range queries with correct serialization semantics requires that conflicts be detected not just at records, but also between records in ranges that are read by a query. Only then can "phantom" inserts into the range be prevented until the range query completes. The RTT approach did not address these conflicts when ordering timestamps.

With locking-based concurrency control, phantom prevention is usually solved through multi-granularity locking with the range as a large granule containing the record as a smaller granule [10]. The range lock blocks the insertion of new records that are not yet in the range and for which it is not possible to hold a record lock. Database systems exploit multi-granularity locking to solve both the phantom problem and to prevent an explosion in the number of locks that need to be maintained. It is not clear how to reconcile the RTT approach with multi-granularity locking.

**2.2.3. Timeslice Queries.** A timeslice query requests the state of part of the database as of some particular time that we call the read time. To correctly support timeslice queries, we must schedule transactions executing timeslice queries correctly in the transaction serialization order. These requirements do not differ from the requirements we normally place on transactions.

What makes this task different from what we have discussed to this point is our desire to execute timeslice queries, which may be only part of a larger transaction, *without locking* the data that they read. This requirement stems once again from our goal of compatibly extending existing database systems, some of which currently execute snapshot transactions without locking. How to do so in the presence of CURRENT requests has not been considered before, including in the RTT approach.

## 2.3. Our Contribution

Compatible extension to existing database systems applies not just to functionality as discussed above. It also involves a desire to evolve current database implementations to provide that functionality. For this it is surely convenient if we can localize the code responsible for timestamp functionality within the database system code base. It turns out that this is possible. We enhance the lock manager present in almost all the database systems and already correctly serializing transactions. We term this augmented lock manager a *timestamping lock manager* (TLM). The TLM provides bounds on a transaction's timestamp that constrain it to agree with the serialization order that it already provides. Logically, the TLM assigns a timestamp to all transactions, independently of whether or not they access transaction-time data.

Performance is always an important issue. We focus on exactly when it is necessary to check timestamp bounds. In particular, we identify several situations in which checks are avoidable. We also identify when we do not have to maintain the information needed for checking timestamps. For example, if no transaction has asked for CURRENT_TIME, we do not need to check timestamp information, and, at least in one strategy, do not even have to maintain this information. In summary, our timestamping lock manager enables more sophisticated locking strategies, such as multi-granularity locking, along with important refinements that offer better performance.

In Section 3, we describe the previously proposed RTT approach, as a basis for the approach proposed here. Section 4 presents our timestamping lock manager in its basic form. Section 5 describes our strategies to reduce the overhead added to the TLM for timestamping, by identifying when checking is not needed, and when maintaining auxiliary information is not needed. Section 6 shows how we can gracefully move from a strategy that minimizes overhead to one that minimizes the frequency of aborts. Section 7 discusses time-slice queries and snapshot serializability, and shows how the TLM can provide this without locking. A final section provides a short summary and discussion.

## 3. The RTT Approach

The RTT approach [9] orders transaction timestamps so as to agree with the serialization order of the transactions when only transaction-time tables are supported.

### 3.1. Bounds on Timestamps

To minimize aborts, the RTT approach supports a flexible choice of the timestamp $T_A$ of a transaction $A$. The approach maintains a *lower bound* $LB_A$ and an *upper bound* $UB_A$ for the timestamp values that can be assigned to $A$. As long as this open-closed interval is non-empty ($LB_A < UB_A$), a legal timestamp assignment exists. At commit, a transaction is assigned a timestamp that is one chronon larger than its lower bound. If the interval becomes empty, transaction $A$ cannot commit, and it is aborted.

When transaction $A$ starts, $LB_A$ is set to the current time and $UB_A$ is set to the largest possible time value. Lists that record the data items read, inserted, and deleted by the transaction are initialized as being empty. These lists are used for post processing at commit time. For each data item $d$, a variable $d.\mathrm{T}^{\mathrm{R}}$ is introduced that records the largest timestamp among transactions that have read $d$.

The RTT approach maintains $LB_A$ to ensure that $d.\mathrm{TT}^{\vdash}$ of any item $d$ read by $A$ and that $d.\mathrm{T}^{\mathrm{R}}$ and $d.\mathrm{TT}^{\vdash}$ of any item $d$ written by $A$ will be earlier than $T_A$, the timestamp we will assign to $A$. When $A$ reads a data item $d$, $LB_A$ is set to $d.\mathrm{TT}^{\vdash}$ if this increases the bound. When it writes an item $d$, $LB_A$ is set to the maximum of its current value, $d.\mathrm{T}^{\mathrm{R}}$, and $d.\mathrm{TT}^{\vdash}$. At commit, the read, insert, and delete lists are processed. We set $d.\mathrm{T}^{\mathrm{R}}$ to $T_A$ for all data items $d$ read by $A$ when this increases the value of $d.\mathrm{T}^{\mathrm{R}}$. Each $d$ inserted or deleted in $A$ is timestamped with $T_A$.

The RTT objective was to ensure that a transaction's timestamp is consistent with the value of "now" that the transaction sees in its requests for CURRENT_DATE, CURRENT_TIME, and CURRENT_TIMESTAMP (in standard SQL; other time granularities are supported in non-standard ways in specific DBMSs [17]). The RTT approach constrains the upper bound $UB_A$ and lower bound $LB_A$ of $A$ as a result of these requests. For example, if CURRENT_DATE is requested, $LB_A$ is set to the maximum of its current value and the first chronon during the date returned and $UB_A$ is set to the minimum of the last chronon of the date returned and its current value. This procedure exploits that CURRENT results have coarser granularities than transaction timestamps. For example, a transaction's request of CURRENT_DATE yields an interval for the timestamp of possibly the entire day; a subsequent CURRENT_TIME request reduces that interval to at most one second. Requests for "now" at any granularity are thus supported.

The RTT approach uses start and stop times of each data item to ensure that write-write (WW) and write-read (WR) conflicts are handled correctly. For read-write (RW) conflicts, it remembers $d.\mathrm{T}^{\mathrm{R}}$ for each data item $d$, which records the last time the item is read. Conservative approximations of these values are kept in a *read-timestamp table* (RTT, hence the name of the approach) that does not retain information about each item, but rather identifies item *classes* by means of a hash function used to index the table. We apply a hash function to data items that distributes them among some number, e.g., 512, of classes. For each such class, the RTT then records a time that is no smaller than the largest $d.\mathrm{T}^{\mathrm{R}}$ for the data items $d$ that hash to that class. This arrangement preserves correctness and enables efficient management of read timestamps.

The advantage of this approach is that the size of the RTT can be limited without requiring a garbage collection scheme were this information to be retained (at least initially) for each data item. The drawback of associating times with data record classes rather than with the records individually is that this approach may result in additional aborts. However, by adjusting the range of the hash function (and hence the size of the table), one can control the trade-off between these two costs (memory costs and increased aborts).

### 3.2. Limitations

The RTT approach falls short in three respects. First, it fails to accommodate non-temporal data. It simply assumes that all data contain the timestamp $d.\mathrm{TT}^{\vdash}$. Second, it assumes record granularity locking; it is not clear how to generalize RTT to multi-granularity locking, which is required to prevent phantoms. Third, it is not clear how to accommodate timeslice queries exploiting snapshot isolation that avoid locking.

We tested the RTT approach in a prototype based on Berkeley DB [16], which provides (uni-granularity) page locking in its B-tree access method. We needed to extend the RTT approach for this to work.

**1.** We had to introduce a *write-timestamp table* (WTT) that is only used for non-temporal data. This table is analogous to the RTT and stores the write times of non-temporal data items. The table enabled us to deal with WW and WR conflicts for the non-temporal data.

**2.** We had to change the entries in the RTT from data items to data pages. Berkeley DB solves the phantom problem by page locking (a form of range locking). By remembering times associated with pages, we are able to correctly compute transaction times that are consistent with serialization order.

This did have some negative consequences, however.

1. Both RTT and WTT record timestamps at the granularity of pages. Thus, the conflict induced timestamp order is very conservative, and that can lead to excessive aborts.
2. The solution is very specific to Berkeley DB. It is not clear how to apply this approach to other locking protocols, in particular to systems that exploit multi-granularity locking to reduce locking conflicts. Small granularities reduce timestamp ordering constraints and lead to fewer timestamp induced aborts.

What we want is an approach that can apply to a wide class of database systems, where it is obvious that the serialization order and timestamp order agree, where execution cost is low, and where sophisticated conflict reduction techniques such as multi-granularity locking can be applied. We consequently pursued an architecture that localizes the timestamp management code and integrates it with the lock manager.

## 4. Timestamping Lock Manager

The timestamping lock manager (TLM) fully supports multi-granularity locking while also enabling CURRENT requests and accesses to both temporal and non-temporal data. Just as important, this approach in many situations avoids some or all of the checking when it can be determined a priori that such checking is not needed. We present the fundamentals of this approach and examine a spectrum of refinements. Selecting the refinements most appropriate for an actual DBMS depends strongly on the specifics of that DBMS's lock manager, which is an intricate and highly optimized piece of code. Similarly, the detailed performance improvements possible with each refinement also depend strongly on those specifics.

Fundamentally, the TLM needs to ensure that each transaction has a time later than the times of all earlier conflicting transactions. That is, our current transaction must be later in time than the transactions that have made earlier accesses in conflicting lock modes to its accessed resources. Enforcing this within the lock manager can immediately be seen to guarantee that serialization and timestamp orders agree. Thus, the latest timestamp of any earlier conflicting transaction (the conflict modes are specified by the lock manager) becomes a lower bound for a transaction timestamp, i.e., earlier times are not acceptable because then timestamp order would not be consistent with conflict order. CURRENT requests are handled the same way as with the RTT approach.

Our TLM, in addition to the normal functions of a classical lock manager, maintains lower and upper bounds for each transaction. It checks these bounds and aborts transactions for which it is not possible to assign a correct timestamp, i.e., when a timestamp range is empty. We describe here how we determine the bounds, and how these are maintained and checked during TLM execution.

### 4.1. The Access Timestamp Table

The TLM inspects conflicts to determine $LB_A$ for each transaction $A$. The TLM extends a conventional lock manager with information on timestamps for preceding conflicting accesses. We define an *access timestamp table* (ATT). An ATT entry, like an RTT entry, contains the timestamp of the last access to any of a set of resources (assigned via a hash function). Unlike the RTT, it does this for each lock mode supported by the lock manager. Thus, the ATT supports the normal intention locks used to provide multi-granularity locking. Further, the ATT maintains this information for each resource locked by the lock manager. Thus, the ATT includes the same range resources that are used by the lock manager to prevent phantoms. Like the RTT, the ATT needn't persist across crashes since transactions after a crash will have timestamps later than the time of the crash.

The ATT stores, for each lock mode and for each entry $i$, the largest timestamp of any earlier committed transaction accessing a resource that hashes to $i$ (i.e., $h(R) = i$) with an access in that lock mode. We then define our lower timestamp bound for transaction $A$ as follows.

$$LB_A = max\{\mathrm{ATT}(h(R).\ m) \mid \text{transaction } A \text{ accesses resource } R \text{ in mode } m_A \text{ and } m_A \text{ conflicts with } m\}$$

The TLM computes the lower bound incrementally. When a transaction accesses a resource, it first acquires a lock in a mode $m_A$ appropriate for the access. When the lock is granted, the TLM updates $LB_A$ by examining the entry for the resource for each mode that conflicts with $m_A$. The TLM then checks whether $LB_A < UB_A$. If not, then the TLM aborts the transaction. The TLM is illustrated in Figure 1. Note that it is possible to implement most of the TLM as a wrapper around the usual database lock manager.

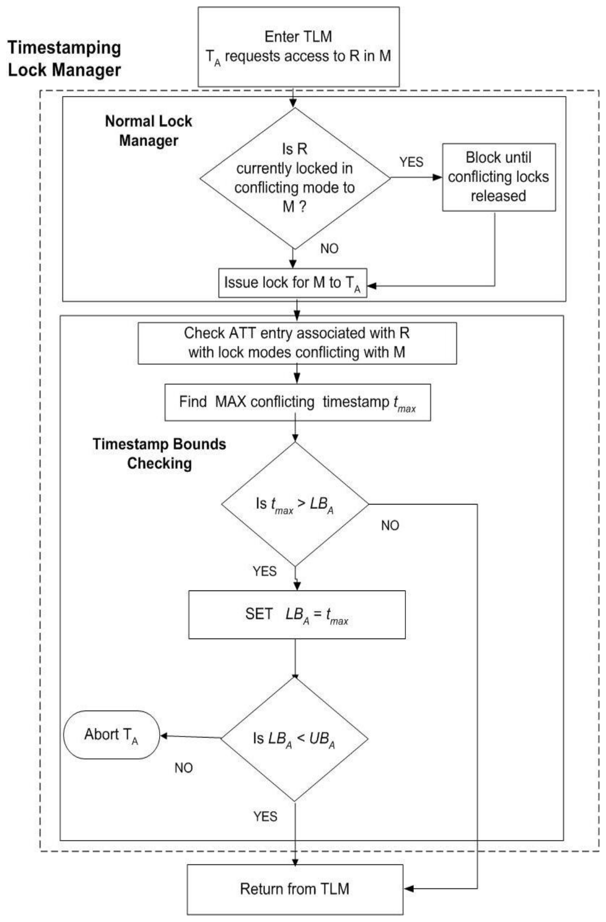
Timestamping Lock Manager
Enter TLM
$T_A$ requests access to R in M
Normal Lock Manager
Is R currently locked in conflicting mode to M ?
YES
Block until conflicting locks released
NO
Issue lock for M to $T_A$
Check ATT entry associated with R with lock modes conflicting with M
Find MAX conflicting timestamp $t_{max}$
Timestamp Bounds Checking
Is $t_{max} > LB_A$
NO
YES
SET $LB_A = t_{max}$
Abort $T_A$
Is $LB_A < UB_A$
NO
YES
Return from TLM

**Figure 1. Timestamping Lock Manager**

## 4.2. Using *d*.TT$^{\vdash}$ Instead of the ATT

In a transaction-time database, the current version of record $d$ contains the timestamp of the record's last writer as $d$.TT$^{\vdash}$. So we can avoid storing a transaction time for $d$ in the ATT and derive it instead from $d$ itself when $d$ is a transaction-time item. The item $d$ is the precise resource, while the ATT entry identifies a resource class. So using $d$.TT$^{\vdash}$ provides a more refined result. We then avoid updating the ATT when dealing with write locks on $d$.

But we do have to check the timestamps in the records that we write, instead of checking the ATT entries. This is like the RTT checking, where we only stored read lock information in the RTT. Whether we choose to use $d$.TT$^{\vdash}$ or the ATT is a matter of implementation convenience. We might proceed as if $d$ did not have a $d$.TT$^{\vdash}$, treating all data the same. This convenience comes at a price, however, because the coarser resource classes represented by the ATT may result in a greater number of aborts.

## 4.3. Commit Time Actions

At transaction commit, the system assigns a timestamp to the transaction in the open-closed interval $(LB, UB]$. This permits the transaction to successfully commit. If each transaction always checks the times for conflicting earlier accesses in the ATT at the time of each access, and all transactions update the ATT when committing, we minimize the time that we can assign as a transaction's timestamp. ATT entries are monotonically increasing. Early checking means that the ATT entry seen will be less than or equal to any later value. Not updating the ATT until commit delays the time at which an ATT entry increases.

Choosing the timestamp to be as early as possible means that a transaction $B$, reading or writing a resource previously locked by $A$, is less likely to have an empty timestamp range. An empty range means that B has to abort to maintain consistency between serialization order and timestamp order.

We need to maintain the ATT so that it continues to contain the latest timestamps of earlier committed transactions that accessed data using the various lock modes. Since we assume strict two-phase locking for transaction isolation, a transaction cannot gain access to a resource until prior conflicting transactions release their locks. Thus, at commit of transaction $A$, for every resource $R$ locked by $A$ in a mode $m$, we post $T_A$ in ATT($h(R).m$) if that time exceeds the current value in that entry.

## 5. Reducing Overhead

A big attraction of deferring timestamp choice until transaction commit is that it is very inexpensive, requiring

- no accessing of extra information while the transaction is accessing data, and
- no extra updating at commit of auxiliary information used solely to correctly maintain timestamp order.

However, a second data access to post timestamps to each updated data item is required.

Unfortunately, this classical technique in its pure form cannot deal with CURRENT requests. Nonetheless, we want to drive our costs closer to commit time timestamp choice.

There are three costs incurred by the TLM approach for dealing with CURRENT requests.

1. The TLM needs to maintain a lower bound $LB_A$ and an upper bound $UB_A$ for each transaction $A$, independently of whether or not $A$ accesses transaction-time data.
2. The TLM needs to check, whenever a lock is granted to transaction $A$, that $LB_A < UB_A$.
3. The TLM maintains the ATT to compute the lower bound on every lock acquisition. This is needed to permit us to choose a timestamp for the transaction to be as early as possible.

If we can accept a slightly higher risk of aborting by choosing a transaction's timestamp to be later, some large efficiency improvements are possible. The key assumptions are as follows.

**A1.** We choose $T_A = min\{t_{Acom}, UB_A\}$, where $t_{Acom}$ is the time when transaction $A$ commits. (Recall that the RTT and prior TLM approaches both set $T_A$ to $LB_A$ + one chronon.)

**A2.** If $A$'s CURRENT request occurs before $UB_A$, then $UB_A$ can be moved earlier, but not earlier than when the request was made. If $A$'s CURRENT request occurs after $UB_A$, then $UB_A$ is unchanged. Thus $UB_A$ is never moved earlier than $t_{cur}$, the "current time".

In concert, these assumptions reduce the need both to check and to update the ATT. Indeed, we can exploit them to avoid maintaining an explicit lower bound. Implicitly, the lower bound becomes the earlier of $t_{cur}$ or $UB$.

### 5.1. Reducing ATT Checking

Transaction $A$ checks the ATT to ensure that there is a feasible timestamp consistent with the access conflicts. We can bypass this check of the ATT when $t_{cur} < UB_A$.

1. No earlier committed transaction can have a time later than $t_{cur}$ by A1, which requires that earlier committed transactions have earlier times.
2. Any transaction that commits later and conflicts with $A$ will have a timestamp later than $t_{Acom}$.
   a. A non-checking transaction, that is, one whose upper bound is later than $t_{cur}$, will have a later timestamp by A1, which requires that its timestamp be the time that it commits.
   b. A checking transaction (with $UB \le t_{cur}$) will, by checking, be later than prior conflicting transactions. By A2, $UB$ is never moved earlier than the commit times of transactions whose conflicts are already checked.

Hence, a transaction $A$ does not have to check that $LB_A < UB_A$ and that $A$ needs to abort until $t_{cur}$ is later than $UB_A$. Checking transactions check that conflicting entries in the ATT have timestamps less than $UB$, and abort when the check fails. Thus, no transactions need maintain $LB$.

In this modification of the TLM approach, some transactions have $UB < t_{cur}$ and are checking the ATT, some transactions have $UB > t_{cur}$ and are not checking, and some transactions have not yet made a CURRENT request, and so are not checking either, because their $UB$ is at "forever."

### 5.2. Reducing ATT Maintenance

If *no* active transaction is checking the ATT, a further optimization is possible: committing transactions do not have to update the ATT. The reason is that all committed transactions (and hence all earlier, conflicting transactions) have timestamps earlier than $t_{cur}$. Hence, conflicts with these earlier transactions will not violate the required agreement between serialization order and timestamp order, since non-checking transactions always commit at $t_{cur}$. Further, if we start maintaining the ATT when we start checking, all transactions that commit after we start checking will have timestamps later than the value of $t_{cur}$ at the time we started to check. But these timestamps will be in the ATT, and our checking will discover the conflicts and order the timestamps correctly.

We describe below two ways of exploiting these observations.

**5.2.1. "Current Request" Strategy.** In the Current Request (CR) approach, a transaction starts checking the ATT when it makes a CURRENT request. Before that, it has no upper bound, and hence the upper bound is not earlier than tcur. When transaction A makes a CURRENT request, the value of LBA is set to the maximum of $t_{cur}$ and first chronon (say the first microsecond) of the requested granularity (e.g., day for CURRENT_DATE, second for CURRENT_TIME), and the upper bound UBA is set to the last chronon of the requested granularity.

Additionally, we do not maintain the ATT unless there is an active transaction that has made a CURRENT request and hence is checking the ATT. This is clearly sufficient as we begin checking even before we reach the upper bound. If no active transaction has requested CURRENT, then committing transactions do not have to update the ATT.

The bookkeeping required is very simple. When transaction *A* first makes a CURRENT request, it becomes a checking transaction. It then increments a "*checking transactions*" counter *CN*. When *A* commits or aborts, it decrements *CN*. A committing transaction checks *CN*, and only posts its timestamp to the ATT when $CN > 0$. A checking transaction decrements *CN* during its commit prior to checking the counter. Thus, if it is the only current checking transaction, it need not update the ATT.

**5.2.2. "Upper Bound Checking" Strategy.** The Upper Bound Checking (UBC) approach checks the ATT even less frequently than in the CR approach, by only starting to check precisely when UB $\leq$ $t_{cur}$. Checking is less frequent than in the CR approach because a CURRENT request that would trigger checking in CR might have an upper bound not yet exceeded by $t_{cur}$ and so this CURRENT request would not invoke checking.

As before, there is no need to maintain the ATT until there is a transaction that will check it. Thus, we delay updating the ATT until at least one transaction *A* exists with $UB_A \leq t_{cur}$. However, because transactions may become checking transactions via the mere passage of time as opposed to performing some action themselves, transactions cannot register themselves as checking transactions. The ATT must be immediately maintained once this condition is satisfied.

To "register" transactions as checking transactions, based on their *upper bounds*, the system maintains a check list *CL* of the transactions that have made CURRENT requests. Each *CL* element is a pair $<I_A, UB_A>$, where $I_A$ is the transaction id for transaction *A*. We add this pair to *CL* as soon as *A* has an upper bound, i.e., as part of its first CURRENT request. *CL* is ordered by the transactions' *upper bounds* (this can be done efficiently by maintaining *CL* as a priority queue). We remove *A*'s *CL* entry when *A* commits. Because we enter a transaction on *CL* as soon as it makes a CURRENT request, *CL* exists prior to our needing to maintain the ATT.

We don't begin updating the ATT until the earliest upper bound on *CL* is reached. However, note that the transaction with the earliest upper bound, i.e., the one that triggers the updating, does not itself need to update the ATT with its commit time. No subsequent transaction can commit with an earlier time and so no timestamp ordering violation is possible. We stop maintaining the ATT when no checking transaction is active, as with the *CR* approach, but with fewer checking transactions.

## 5.3. Performance

In this section, we examine the performance of the TLM. First, note that database concurrency is unaffected by the timestamping, as locking conflicts are unchanged. Also concurrency can be substantially less constrained than in Berkeley DB with its page granularity. The TLM exploits the granularity supported by the original DBMS. We characterize the locking overhead involved in the completely unoptimized case, and then consider the impact that our optimizations have on reducing overhead. We argue that the overheads involved for expected system behavior are not large, especially when our optimizations are exploited.

The performance improvements possible with each refinement are very specific to the lock manager being extended and to system workload: such things as number of locks, conflict rate, lock manager path length, etc. Hence, we make only general observations.

First, what is the impact of the lock manager on execution path length seen by transactions in a database system? There is no one specific figure, of course, but the concurrency control and recovery subsystem in a transaction processing application is typically 5–10% of the path. TP applications typically do more locking than most applications, so this is a high figure. The lock manager is less than half of that. Thus, at most 5% of the path is in the lock manager. So timestamping will impact total application path by at most a few percent.

According to Gray and Reuter [7], lock manager instruction path for a non-blocking lock request is a few hundreds to one thousand instructions. This includes call overhead, searching lock lists, and testing their resource ids and lock modes. Releasing locks is usually done *en masse* at transaction end, incurring call overhead only once for all of a transaction's locks.

The incremental impact of timestamping is surely less than 20% to 30% of the path length of a lock manager without timestamping (equivalent to 1% to 1.5% of the full execution path length). Checking timestamps in the ATT involves no list searching. Rather, a hash is computed (a few instructions) and ATT entries are checked or updated. Checking and updating together examine each lock mode of a locked resource. This may be as many as five or six comparisons and/or updates. The extra instructions are not more than a hundred. Further, as explained above, checking or updating the ATT is often unnecessary.

The above overheads are not trivial, but given the modest contribution of the lock manager to transaction execution path, the impact on system performance will not be more than 2–3%. And our optimizations frequently eliminate the great bulk of this extra overhead. When all temporal transaction timestamps can be at their commit times, ATT checking during lock requests and ATT updating at commit are avoided, as is true when no temporal transactions are executing. In those cases, the extra lock manager overhead to check is minimal.

We have only discussed the extra TLM cost for calculating the time used in a timestamp. We have not discussed storing timestamps in versions, which is more costly. We believe that this timestamping should be done after commit, as recommended by Salzberg [14]. Then the timestamping is outside of the response time for the transaction, and can be combined with page accesses that are already required for other reasons, greatly reducing this overhead.

## 6. Controlling the Abort Rate

The optimizations of Section 5 require that the timestamp we assign be either the *upper bound* for a transaction or $t_{cur}$ at the time of commit, whichever is earlier. This assignment is correct and also minimizes the TLM checking. However, it can increase the aborts required to keep timestamps consistent with serialization order. This might be a problem if database systems reply to CURRENT requests with a very precise time (e.g., a request for CURRENT_TIMESTAMP at a precision of micro-second). Here, we explore both the nature of the problem and how we can extend our approach to minimize this problem.

### 6.1. Increased Aborts

Suppose transaction *A* starts executing at 1 P.M. and asks for CURRENT_DATE at 11 P.M. *A* will have a timestamp, assigned by either Current Request or Upper Bound Checking approach, not earlier than 11 P.M. Another method might have been able to give *A* a timestamp as early as 1 P.M. While *A* may itself be able to commit, another transaction *B*, which reads data written by *A*, may have an *upper bound* of, say, 2:15 P.M., by virtue of requesting CURRENT_TIME at that minute. *B* could have committed had *A* gotten the earlier timestamp. But with a timestamp after 11 P.M. for *A*, *B* must now abort.

While abort frequency can increase, it may still be small.

1. The number of distinct resource classes, i.e., the range of the hash function, can be made large, reducing the chance that a subsequent transaction encounters the effects of a long-running transaction as in our example.
2. Most requests for CURRENT_TIME result in much smaller intervals between the earliest possible correct timestamp assignment and our technique of assigning the latest possible timestamp. CURRENT_TIME results in at most one second between the lower and the upper bound. Thus the flexibility of the more conservative approaches to keeping track of bounds for timestamps will be less than the example suggests.

Regardless of the absolute number of additional aborts, we would still like to be able to reduce the number of aborts while providing much of the optimization gain of reduced ATT checking and updating.

### 6.2. Earlier Transaction Timestamps

The earlier a transaction's timestamp, the fewer other transactions need be aborted to preserve consistency between timestamp and serialization orders. To reduce aborts we need to maintain an explicit *lower bound* for transaction timestamps in addition to an *upper bound.* We then can choose a transaction's timestamp that is at the *lower bound* of its timestamp range. And if we want it to be able to have a timestamp that can be earlier than the time at which we begin checking, we need to start maintaining the ATT even earlier.

We do not know whether a transaction timestamp earlier than $t_{cur}$ at commit is legitimate unless we maintain the ATT and check a transaction's lock request against up-to-date ATT entries. Because no transaction that committed earlier than $t_{check}$, the time at which we begin checking and maintaining the ATT, can have a timestamp later than $t_{check}$, *we can begin checking at any time*, and use $t_{check}$ as the initial *lower bound* for currently active checking transactions.

So if we are incurring too many aborts, we begin updating and checking the ATT. This can be done at our convenience! We call this the *Adaptive Strategy*. When we want a transaction with a timestamp earlier than $t_{cur}$ at commit, we make it a checking transaction. The initial lower bound for any transaction is then the maximum of its start time and $t_{check}$. We maintain upper bound as before.

We can provide a *lower bound* selectively, exploiting our prior optimizations. That is, not all transactions need to be checking transactions, and such non-checking transactions do not need to maintain a lower bound. But once the transaction is a checking transaction, we maintain a *lower bound* as well as an *upper bound* for transaction timestamps; and we choose our timestamp to be at the *lower bound* of this range.

Should we continue to have too many aborts, we can make all transactions checking transactions.

## 7. Lockless Timeslice Queries

We timestamp data to support transaction-time database functionality, an important part of which is timeslice queries. A *timeslice query* requests the state of part of the database as of the query's *read time*. A transaction with snapshot isolation typically queries the database with a read time equal to the transaction's start time. With full transaction-time database support, a transaction can query the database as of any past time. The result should be a transaction-consistent view of the database with versions of data items read that have the largest timestamps less than or equal to the *read time*.

Our desire is to execute timeslice queries, which may be only parts of larger transactions, *without locking* the data that is read. An important aspect of snapshot queries in several commercial databases is that they are executed without locking. Indeed, the point of snapshot isolation is to enable snapshot reads without locking, so that readers are never blocked by updaters, and updaters are never blocked by snapshots. So this is important for compatibility.

### 7.1. Impact of Timeslice Queries

What happens when a timeslice query in transaction *B* reads data updated by a transaction *A* that is active during the same time that *B* does its read? Let *A's* bounds on its timestamp be $LB_A$ and $UB_A$, and *B's* time of read ("as of" time) be $R_B$. There isn't a concern when *A* commits before *B* reads the updated data, as then we have access to *A's* timestamped updates. But it is a concern if *B* reads this data before *A* is committed. There are two cases, which we describe next: (i) timeslice reader accesses the data first and (ii) updater accesses it first.

**7.1.1. Timeslice Reader First.** When timeslice reader *B* reads data before updater *A* does its update, we need to make *A* aware of $R_B$. Thus we need *B* (i) to assign resource identifiers to the data that it reads that are the same resource identifiers used for locking data and (ii) to have *B* update the appropriate ATT lock mode entries with $R_B$. The appropriate lock mode will usually be S or shared, the mode used had *B* been an ordinary transaction reading the data. B's read data can be accessed immediately by conflicting operations because a timeslice query does no locking. Hence *B* must update the ATT immediately. *A*, which will be a "checking" transaction, looks at entries in the ATT and checks whether there are timestamps for conflicting operations that are later than $UB_A$. If so, *A* must abort. Otherwise, if later than $LB_A$, then $LB_A$ must be increased to this later time, just as if the timeslice query had been a read by a committed transaction.

**7.1.2. Updater First.** To deal with uncommitted updates to the data that it reads, timeslice reader *B* checks for locks that would conflict with its read were it a normal reader. If it finds such a lock, then it must check the timestamp bounds of all transactions *A* holding the lock. If all $UB_A$ are later than $R_B$, then the read proceeds without blocking and, as in the "timeslice reader first" case, we update the ATT to inform future updaters of *B*'s read time. And if $R_B$ is later than $LB_A$, then $LB_A$ is increased to this later time. If *B* finds an uncommitted update from some *A,* where $UB_A$ is earlier than $R_B$, then to avoid blocking timeslice query *B*, we must abort either *B* or *A*. Aborting updater *A* keeps the story simple for timeslice queries, i.e., they run safely without blocking, locking, or aborting. We expect this is the alternative most implementations will choose.

## 7.2. Handling Timeslices with the TLM

The preceding discussion suggests a way of dealing with the interactions between timeslice queries and updaters within the TLM. We have *B's* timeslice query call the TLM like an ordinary query, and update the ATT as required. But the TLM makes some adjustments for "timeslice locking", e.g., it never blocks timeslice queries.

For *B's* timeslice query, instead of waiting until *B* commits to post timestamps to the ATT, the TLM immediately updates the S mode entry for the appropriate resource in the ATT with $R_B$. Instead of blocking a timeslice read when this resource is held in a conflicting lock mode, the TLM permits the read to proceed. When the conflicting lock is held by updater *A* with $UB_A$ earlier than $R_B$, *A* is aborted. Otherwise, if $LB_A$ is earlier than $R_B$, we set $LB_A$ to $R_B$, ensuring that *A's* timestamp is later than $R_B$.

The locking overhead of timeslice queries can be minimized by exploiting "large grained" resources when a timeslice query *checks* locks and *updates* ATT entries. A trade-off exists between this overhead and the number of update transactions that might be impacted. As an example, let our timeslice "granule" be a relational table. A conflict is manifested by the fact that the IX table lock of updater *A* conflicts with the S table "timeslice lock" of *B*'s timeslice query. If $UB_A$ is earlier than $R_B$, then *A* is aborted.

## 7.3. Optimizing Timeslice Queries

Sometimes non-locking timeslice queries are easy to realize and have no impact on the timestamping of remaining transactions. We can then avoid the checking and updating of the ATT. This occurs when we know that read time is earlier than the earliest lower bound for any uncommitted transaction. Such a timeslice query cannot be impacted by uncommitted transactions, as such transactions cannot provide data with a timestamp that is early enough to be seen by the query. Further, the query has no impact on the timestamps of uncommitted transactions, since all these transactions will have timestamps later than its read time. Any updater that will commit later than $t_{cur}$ will never be aborted, since timeslice read times will always be earlier than $t_{cur}$. Thus, if no transaction has made a CURRENT request, then no access to the TLM is required to check or update the ATT. This is why, in a conventional DBMS, a snapshot query safely executes without locking. Its read time is *always earlier* than the possible commit time of *any active transaction*.

## 8. Summary

The only prior timestamping work that supports CURRENT requests is the RTT approach [9]. It supports neither access to non-temporal data nor multi-granularity locks. As mentioned in Section 2, we had to extend the RTT approach in our Berkeley DB prototype to support access to non-temporal data, and even then the page-grained locks would produce additional aborts. Furthermore, the RTT approach requires careful, ad hoc integration with the existing concurrency control facilities of the DBMS.

This paper introduces a compatible extension of a DBMS to realize temporal functionality. The existing lock manager is augmented to maintain a lower and an upper bound for each transaction. The resulting Timestamping Lock Manager (TLM) utilizes an *access timestamp table* (ATT), which contains the timestamp of the last access to each resource class for each lock mode supported by the lock manager. The beauty of the TLM approach, and the variants presented here, is that if the lock manager correctly serializes transactions then the timestamps will agree with that order. This includes locking approaches to the phantom problem, such as key range locking.

The basic TLM imposes runtime overhead for checking and maintaining the ATT for all queries, even when this checking is not strictly needed. Thus we described ways to optimize the TLM to avoid this. We showed that full ATT checking and updating can start at any time, allowing flexible and dynamic trade-offs: earlier checking, which increases CPU overhead, but decreases the number of timestamp-induced aborts versus later checking, which decreases overhead, but may increase aborts.

We also considered timeslice queries, proposing ways to handle these without locking that nonetheless work with variants of the TLM approach. The proposed mechanisms check and update the ATT but do not require locking data read by the timeslice query, ensuring that a timeslice query is never blocked. Also discussed was to require that a timeslice query read time be earlier than the possible commit times of active transactions, which then avoids the TLM mechanisms.

The result is a collection of approaches that support: (i) timestamping at commit, (ii) serialization-consistent timestamps, (iii) CURRENT requests whose responses are consistent with transaction timestamps, (iv) accessing non-temporal data, (v) range locking and multi-granularity locking, and (vi) non-locking timeslice queries.

## 9. References

[1] Atempo, Inc., **Time Navigator**, 2004.

[2] J. Bair, M. H. Böhlen, C. S. Jensen, and R. T. Snodgrass, "Notions of Upward Compatibility of Temporal Query Languages", *Wirtschaftinformatik* 39(1), 1997, pp. 25–34.

[3] C. Bettini, C. E. Dyreson, W. S. Evans, R. T. Snodgrass, and X. S. Wang, "A Glossary of Time Granularity Concepts," in *Temporal Databases: Research and Practice*, O. Etzion, S. Jajodia, and S. Sripada (eds), Springer-Verlag, pp. 406–413, 1998.

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

[5] C. Bettini, S. Jajodia, and S. X. Wang, *Time Granularities in Databases, Data Mining and Temporal Reasoning*. Berlin, Springer-Verlag, 2000.

[6] 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), 1997, pp. 171–214.

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

[8] C. S. Jensen and C. E. Dyreson (eds), "A Consensus Glossary of Temporal Database Concepts—February 1998 Version", in *Temporal Databases: Research and Practice,* O. Etzion, S. Jajodia, and S. Sripada (eds), Springer-Verlag, 1998, pp. 367–405.

[9] C. S. Jensen and D. B. Lomet: "Transaction Timestamping in (Temporal) Databases", *VLDB,* Rome, 2001, pp. 441–450.

[10] D. B. Lomet, "Key Range Locking Strategies for Improved Concurrency", *VLDB,* Dublin, 1993, pp. 655–664.

[11] Lumigent Technologies, Inc., **Log Explorer**, 2004.

[12] J. Melton (ed), ISO/IEC 9075:1992, "Database Language SQL," 1992.

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

[14] B. Salzberg, "Timestamping After Commit", *PDIS,* Austin, 1994, pp. 160–167.

[15] B.-M. Schueler: Update Reconsidered. *IFIP Working Conference on Modelling in Data Base Management Systems,* 1977, pp. 149–164.

[16] Sleepycat Software Inc., **Berkeley DB**, 2001.

[17] R. T. Snodgrass: *Developing Time-Oriented Database Applications in SQL*, Morgan Kaufmann, 1999.

[18] M. Stonebraker, "The Design of the POSTGRES Storage System", *VLDB,* Brighton, 1987, pp. 289–300.

[19] K. Torp, C. S. Jensen, and R. T. Snodgrass, "Modification Semantics in Now-Relative Databases", *Information Systems* 29(78), 2004, pp. 653–683.

[20] R. Weiss, "How Oracle Database 10G Revolutionizes Availability and Enables the Grid", Technical Paper 40164, Oracle Corporation, 2003.