Consistent Timestamping for Transactions in Distributed Systems

David Lomet

Digital Equipment Corporation
Cambridge Research Lab

CRL 90/3 September 17, 1990

Abstract

Tagging data in a database with timestamps that indicate when data was entered can be very useful. It permits a user to query the database as of some historical time. Further, it permits a user to see a transaction consistent "recent" version of the database without having this transaction interfere with ongoing updates. This support requires that timestamp ordering represent a valid serialization of the transactions. Achieving this in a distributed system is potentially troublesome. This paper suggests that the **two phase commit** protocol messages can be used to establish and distribute a correct timestamp to all transaction cohorts. Refinements permit this concept to deal with heterogenous systems where not all cohorts perform timestamping. Early release of read locks can be supported via bounding the range in which a transaction is permitted to commit.

Keywords: timestamps, multiversion, transactions, commit, distributed systems

1 Introduction

1.1 Uses of Timestamps

Over the last five years, multiversion databases have attracted increasing attention. This has led in a number of directions. Temporal databases have been studied with several notions of time [SNO]. Data is "stamped" with the time of interest, and this timestamp can be queried along with the ordinary data.

Our focus is on **transaction time**. All updates made by a transaction to a database are stamped with the same time. This **timestamp** is stored as an attribute of the data. The order of the timestamps must be a correct serialization of the transactions.

Data that is no longer current can be stored separately from current data. This "historical" data is never updated, and hence could be stored on write-once, read many (WORM) optical disks. Data that is current may continue to be updated. Hence this data can advantageously be on a write-many, read-many magnetic disk. Both POSTGRES [STO] and the time-split B-tree(TSB-tree) [LOM] make this point. A very inexpensive WORM medium, such as optical disks, changes dramatically the functionality/cost trade-off and makes multiversion support interesting for a large number of applications.

Having timestamps with data permits users to query a database AS OF some particular time. With the appropriate support, such a query can provide a transaction consistent view of the database as it existed at the requested time. This is precisely the capability that TSB-trees are tailored for as they cluster data together by time. Such temporal queries can be found in financial applications, medical records, engineering design, etc.

Limited versioning can be found in at least one commercial relational database, Rdb/VMS [JOS]. It is used to support a transaction consistent view of RECENT data (called "snapshot" data). This recent version of data supports read-only requests without any interference with on-going update activity. That is, such a read-only request does not need locking, and hence will not produce lock conflicts that may block or impede updating.

1.2 Providing Transaction Timestamps

Timestamps have a long history as a way of performing concurrency control [BER]. Most of the efforts at using timestamps in this way, however, have not turned up in system implementations, where two phase locking usually reigns supreme. Locking is well understood and has acceptable performance.

A key advantage of locking is that the serialization order for a transaction is "chosen" at the time a transaction commits. Essentially, it is serialized after all transactions whose data it has seen. Timestamping methods impose the serialization at the point when the timestamp is chosen. This is frequently when the transaction starts [BER,REE]. Competing requests for the same data that are out of order result in one or the other of the competing transactions being aborted. This is usually considered to be less robust and less effective than locking. With the choice of timestamp at commit time, the timestamp can be chosen to correctly reflect the serialization that the transaction actually requires.

Early choice of timestamps does have one decided advantage. The timestamp is known to all transaction participants (cohorts) and can be propagated to each new cohort when a request is made for service. Commit time timestamping requires that all cohorts be notified at commit time as to what the transaction's timestamp is. And, each cohort should have a role in deciding what that timestamp will be, so that each cohort can guarantee local agreement of serialization order and timestamp order.

The late choice of transaction time is not a large inconvenience in centralized systems. The timestamp choice is made at a single system node and needn't be propagated anywhere. There are two problems in distributed database systems. A transaction:

1. must choose a timestamp that is satisfactory to all its cohorts, where the requirements of a cohort for transaction time may only be known to itself;
2. needs to propagate the chosen timestamp to all cohorts so that the same timestamp is used everywhere.

1.3 The General Approach

Our approach is to defer the choice of transaction time until commit. To cope with distributed cohorts, information exchanged to choose and distribute transaction time is piggybacked on the two phase commit protocol messages [GRA,LAMS]. Thus, we extend the two phase commit protocol to provide a more general agreement protocol. Not only is it used to agree on and propagate the commit/abort state of the transaction. It is also used to agree on the transaction timestamp. This is done without extra message overhead. (This same basic approach was suggested previously in [HER], as part of an optimistic concurrency control method. The context for our work is different and is outlined in the next paragraph. We also extend the approach to provide increased capability. We were not aware of this prior work during the time that our approach was developed.)

We place this approach in the traditional two phase locking context and show how timestamps can be propagated using the two phase commit protocol in section 2. Section 3 discusses how we can extend the timestamping commit protocol by supplying timestamp ranges. This permits us to exploit common commit protocol optimizations. In section 4, releasing read locks at prepare time is discussed. We show how even transactions with non-two phase locking can be successfully serialized. Section 5 discusses how to handle timestamping in a heterogenous system where not all database systems perform timestamping. Section 6 provides a discussion and assessment of the work.

2 Basic Timestamping Mechanism

2.1 The Two Phase Commit Protocol

We begin by giving an informal description of the two phase commit (2PC) protocol. It has the following steps:

1. A transaction coordinator notifies all parties (cohorts) to a distributed transaction that the transaction is now to be terminated, and hopefully committed. This is the PREPARE message (message one) of the protocol.

2. Each cohort then attempts to become PREPAREd. Essentially, this means making certain that it can guarantee that both the before state of the transaction and the after state are durably stored. Either one of these states can be guaranteed to be installed, depending on whether the transaction COMMITs or ABORTs.
3. Each cohort then **votes** on the disposition of the transaction. If the attempt to prepare fails, or any preceeding step of the transaction fails, a cohort votes ABORT. If the attempt to prepare succeeds, then the cohort votes COMMIT. The cohort notifies the coordinator of its vote by sending a message to it. This is the VOTE message (message two). A cohort that has voted COMMIT is now PREPAREd.
4. When the coordinator has received votes from all the cohorts participating in the transaction, it knows the disposition of the transaction. The coordinator commits the transaction if all cohorts have voted COMMIT. If any cohort has voted ABORT, or the coordinator times out waiting for a cohort's vote, then the coordinator aborts the transaction. The coordinator sends the transaction disposition message (i.e. COMMIT or ABORT) (message three) to all cohorts.
5. When a cohort receives the transaction dispostion message, it terminates the transaction according to its direction, making the transaction state either COMMITTed or ABORTed. If COMMITTed, the after state of the transaction is installed in the database. If ABORTed, the before state is re-installed in the database. The cohort ACKs the disposition message upon stably storing the transaction disposition (message four).

There are a number of multi-phase commit protocols. And the 2PC protocol itself has a number of optimizations to reduce messages. Any protocol in which each cohort sends a message to a coordinator and where the coordinator informs all cohorts of transaction disposition can be used to agree upon a transaction time. The methods below should work with many distributed commit protocols, including, e.g., nested commit (linear) 2PC. We discuss the impact of certain protocol optimizations in section 4 and describe extensions that work with these.

2.2 Choosing a Transaction Timestamp

To select a transaction time, we extend the 2PC protocol by augmenting the information conveyed on two of its messages. Basically, each cohort informs the transaction coordinator of its requirements for transaction time. The coordinator then attempts to find a single time that satisfies all cohort requirements.

When a cohort votes to COMMIT a transaction at message number two, it also conveys its requirements with respect to the choice of a transaction time. The coordinator examines all the requirements and tries to find a transaction time that satisfies all of them. If successful, it propagates, on message number three, to all of the cohorts, both the disposition of the transaction and, if the dispositon is COMMIT, the transaction time chosen. Below, we describe the nature of a cohorts' requirements on transaction time, and how the coordinator reconciles these requirements in selecting a transaction time.

2.2.1 How a Cohort Selects Its Transaction Time Requirement

A cohort must determine, when it receives the notification to begin the commit process (i.e. message number one from the coordinator), a time that is later than the time for any preceding transaction with which it may conflict. A transaction conflicts with a preceding transaction if, for example, it reads data written by the preceding transaction or writes data read by the preceding transaction. In this case, the transaction serializes after the preceding transaction [BER]. Our protocol is designed to ensure that timestamp order agrees with transaction serialization order. Enforcing that transaction time be later than the time of preceding conflicting transactions guarantees that timestamp order and serialization order agree.

We assume in the following that each site has a local clock that is loosely synchronized with a global time source that reflects real world time, e.g. Greenwich Mean Time. Our intent is to assign times to transactions that reflect users' perceptions of when the transactions actually occurred. We combine these local clocks with an adaptation of Lamport clocks [LAMO] to ensure that transaction times are monotonically increasing.

The following, conservative procedure yields the timestamp value. That is, a site (database system at the site) that executes the procedure will generate a time for a transaction at the site that is later than the transaction times of all previously committed transactions upon which the committing transaction conflicts. (Note: All symbols used in equations here and subsequently are defined in Table 1.)

1. A database system maintains a monotonically increasing LAST transaction time. It does this by comparing LAST with the timestamps that it receives for each committed transaction in message three of the commit protocol. Whenever one of these timestamps is later than LAST, LAST is set to the value of this new timestamp. This is the Lamport clock component.

2. A database system that acts as a transaction cohort expresses its transaction time requirement as the EARLIEST time at which the transaction can be permitted to commit. This must be later than the time of any preceding conflicting transaction in that database. When the database receives the PREPARE message from the coordinator and it wants to vote to COMMIT, it votes (at message two) an EARLIEST transaction time that is later than LAST and not earlier than the current clock time. Thus, cohort i votes a time for transaction X of

   $$EARLIEST_i(X) = \max\{LAST_i(X) + \epsilon, CLOCK_i(X)\}$$

   A more agressive alternative is to compute EARLIEST every time that the cohort acquires a lock for the transaction, and to remember this EARLIEST value for the last acquired lock. When the PREPARE message is received, the cohort votes the remembered EARLIEST time. This remembered time must be later than all conflicting earlier transactions when locks are held until commit time.

3. The coordinator can pick a transaction time that is not earlier than the latest EARLIEST time chosen for any cohort. In fact, it is desirable to choose exactly the lastest EARLIEST time voted. This transaction time has the advantage of being the time that satisfies the constraints and that also is the closest such time to the times required by the cohorts. The chosen transaction time is distributed to the transaction cohorts on the transaction disposition message (message three) of the 2PC protocol. Thus, the coordinator chooses a time for transaction X of

   $$TIME(X) = \max\{EARLIEST_i(X) | COHORT_i(X)\}$$

We call the time between a cohort's EARLIEST vote and the commit time of the transaction the PREPAREd-INTERVAL. The result of 2 above is that conflicting transactions at a site will have disjoint PREPAREd-INTERVALs when strict two phase locking is used by the cohort database system. Strict two phase locking requires that all locks be held until commit. Hence, a following transaction is prevented from preparing until the earlier conflicting transactions are committed and release their locks. Disjoint PREPAREd-INTERVALs thus guarantee that a following transaction will have a timestamp that is later than all conflicting transactions that precede it in the serialization order at a site.

LAST must be at least as late as the timestamps of any previously committed transaction at the database. A following transaction at a site will thus vote an EARLIEST time that is later than the commit time of all preceding conflicting transactions at the site. The chosen transaction time will then be later than the times of all transactions with which it may conflict at all sites. This assures that serialization order and timestamp order agree at each cohort. Since serialization order and timestamp order agree locally at each cohort, using a common timestamp ensures that these orders will agree globally for all transactions, local and distributed. This makes it unnecessary to record when each data item was last read, which is frequently necessary in timestamping schemes that choose transaction times at transaction start [BER].

The choice of transaction time in item 3 is the smallest (earliest) time that satisfies the constraints of all cohorts. It minimizes the value of transaction time and hence the values at each database of the variable LAST. Its effect is to keep transaction time closer to the clock time seen at each site. This will improve the correlation between "real" (i.e. clock) time and the time that is used to stamp the data in the database system.

3 A Closed Range of Commit Times

3.1 Divergent Clock Time and Transaction Time

The above timestamping extension to the 2PC protocol has a troublesome limitation. One system node with a substantially faster clock can seriously disrupt the entire distributed system and the transaction times that are chosen. It's late EARLIEST vote will always become the transaction time. This forces transaction time away from clock time at chohorts whose clocks are running correctly. But, if a cohort can commit work at 4:00PM, a user at that location does not expect the transaction to have a timestamp of 10:00PM that evening. The user expects a time which is within no worse than a few minutes, and perhaps only a few seconds of the EARLIEST time supplied by the cohort.

Since it is required that transaction timestamp order agree with transaction serialization order, how does one limit the divergence between clock time and transaction time? The answer is that transactions for which the EARLIEST votes of the cohorts are too far apart can be aborted. The tricky part here is what constitutes "too far apart". This is similar to what constitutes reasonable "timeouts" for messages or locks. Below we suggest a way of dealing with this.

3.2 Voting With a Closed Timestamp Range

One way to establish bounds for how divergent transaction times can be is to ask the cohorts, when they vote their EARLIEST time for the transaction, to also vote a LATEST acceptable time for the transaction. The LATEST time is not required for serializability, but is designed to limit clock and transaction time divergence. The transaction coordinator is required to find a transaction time that is within all the [EARLIEST,LATEST] time ranges voted by each cohort. If the intersection of these ranges is null, the coordinator ABORTs the transaction. A coordinator thus chooses transaction time to be

$$TIME(X) = \min\{\cap\{[EARLIEST_i(X), LATEST_i(X)]\,|COHORT_i(X)\}\}$$

Notice that this agrees with our prior time choice when one interprets the absence of a LATEST choice as a vote for a LATEST of infinity.

A heavily used database may well place more stringent requirements, i.e. vote a smaller range, than a lightly used database. It may need the tight bounds to increase concurrency by reducing the amount of time that the transaction is in doubt. Thus, it is important to provide the option for a database to vote both bounds.

A database on a workstation might be willing to accept almost any timestamp that a host database might agree to during a distributed transaction, so long as transaction time order and transaction serialization order agree. Such a database might not vote a LATEST bound.

It is desirable, of course, to correct a divergent clock because it may be the cause of frequent transaction aborts. It is possible to use the ABORT message itself to inform cohorts of the reason for the abort. In particular, an ABORT message informing cohorts that divergent times caused the abort could prompt cohorts to re-synchronize their local clocks with the global time standard.

3.3 The Read-Only Commit Optimization

A read-only cohort, i.e., one that has no updates, usually does not need to receive the COMMIT message in the 2PC protocol, as it has no activity that it needs to perform as a result. It merely releases its locks when it receives the PREPARE message. This violates strict two phase locking locally. We cannot permit read locks to simply be released at PREPARE time. A subsequent conflicting transaction may access this data and commit with an earlier timestamp, hence making timestamp order different from any valid transaction serialization order.

We must be sure that subsequent transactions that write "unlocked" data are given timestamps later than the transaction that released the locks. Hence, we would perhaps prefer to release these locks only after the time of transaction commit. The problem is how to preserve the read-only optimization when the cohort will never be told, via a COMMIT message, the timestamp of the transaction.

It should be immediate that a read-only cohort, sending its COMMIT vote with a closed timestamp range of [EARLIEST,LATEST], solves this problem. This read-only cohort now knows that the transaction will terminate no later than the time it provided in LATEST. Hence, it can free its locks at LATEST time, without ever knowing, via the COMMIT message, the precise time that the transaction terminated. The LATEST vote ensures that the PREPAREd-INTERVALs of conflicting transactions are disjoint, even without knowing the actual commit time of the transactions. And this assures that timestamp order agrees with serialization order.

3.4 In-Doubt Transaction Read Data

The classic problem with the 2PC protocol is that it is subject to being "blocked" in the case of system failures. In fact, there is no commit protocol that resists blocking in all failure cases. A blocked transaction can make the data used in the transaction unavailable for potentially extended periods of time.

Data unavailability is ameliorated by the fact that data that is only read by a transaction can be unlocked at PREPARE time, when timestamping is not involved. Again, the constraint that timestamping requires, i.e. that two conflicting transactions not be simultaneously prepared, limits our response to blocked transactions.

By voting its cohorts with a closed timestamp range, i.e. [EARLIEST, LATEST], a database can restore its ability to release read locks for a blocked transaction. That is, as with a read-only cohort, it knows that the transaction must terminate no later than the time voted as LATEST. Hence, even in-doubt transactions can release their read locks then. This does not save us from the necessity of retaining the write locks of the transaction, as we still do not know whether to install the after state of the transaction, or re-install its before state. It is the write locks that keep this part of the state inaccessible.

4 Releasing Read Locks at Time of PREPARE

4.1 Another Optimization Denied (Perhaps)

In systems without timestamping requirements, any cohort can release READ locks at PREPARE time, so long as there is no further locking activity in the transaction. This reduces lock holding time, thus increasing concurrency. As before, with timestamping, this cannot be done in this direct way. The problem is not solved solely by providing a LATEST time at which the transaction must terminate. The whole point of releasing read locks at PREPARE time is to make the data so locked available to other transactions **before** the transaction commits. We do not want to hold locks until clock time exceeds LATEST.

The important constraint is not one of preventing other transactions from using the read-locked data after its transaction has PREPAREd. This is harmless, as attested by the fact that, in the absence of timestamping considerations, one could freely access this data. Rather, what is required is that a transaction that modifies this data be required to commit with a transaction time that is later than the commit time of the prior prepared transaction. The general problem here is to keep PREPAREd-INTERVALs disjoint for conflicting transactions so that PREPAREd order becomes COMMITTed order and timestamp order as well. Hence, this problem is one of insuring that a subsequent conflicting transaction votes an EARLIEST time that is later than the LATEST time that is voted by the current transaction.

One possible approach is to FORCE the LAST variable to immediately be set to the LATEST time voted. This is unlikely to be satisfactory, however, because it increases the divergence between clock time and transaction time. Such divergence will lead to unnecessary transaction abort or to user surprise concerning transaction time.

4.2 DELAY Locks

What we would like to provide is a way of making read-only data available to subsequent transactions at PREPAREd time but delay any transaction that uses the data so that it will have a transaction time that is later than the PREPAREd transaction that "released" the data. This can be be done with a new lock called a DELAY lock. The idea of a DELAY lock is as follows. At PREPARE time, a transaction transforms all its read locks to DELAY locks. At commit time, the DELAY locks are also dropped. A DELAY lock does not conflict with any other lock mode. However, if a transaction write-locks data that is DELAY locked, it is not permitted to commit until after the DELAY lock is dropped. This ensures that the timestamp order of transactions agrees with their serialization order.

Another way to make use of DELAY locks is to again remember that their purpose is to force transaction time ordering to agree with serialization order, and it is these timestamps that we are trying to control. This suggests that rather than delaying commit processing, i.e. the 2PC protocol, we instead use the DELAY locks encountered by a transaction to control what a transaction votes as its EARLIEST bound for transaction time.

The idea is to examine DELAY locks still held on data that has been modified by a subsequent transaction at the time that the subsequent transaction initiates its commit processing. The latest time on any of the DELAY locks that it saw (not the delay locks that it may set) helps in establishing the lower bound on its permitted transaction time. Thus, a transaction will vote an EARLIEST time that is later than LAST (the time of the last transaction to commit) and the latest time of all DELAY locks seen by the transaction, and not earlier than clock time. That is,

$$EARLIEST_i(X) = \max\{LAST_i(X) + \epsilon, CLOCK_i(X), \max\{LATEST_i(Y) | CONFLICTS_i(Y, X)\}\}$$

This ensures that conflicting transactions continue to have disjoint PREPAREd-INTERVALs, and hence that timestamp order and serialization order agree.

4.3 Implementing DELAY Locks

A low cost way to implement DELAY locks does not involve any explicit downgrading of locks in the lock manager and hence no extra call to the lock manager. Rather, a transaction's read locks needn't be changed and can be explicitly released only at transaction commit. A subsequent transaction that encounters a read lock (and that wishes to write the data so locked) consults the transaction table to determine the disposition of the transaction.

If a transaction holding a READ lock is PREPAREd, the READ lock is treated as a DELAY lock, and a requested WRITE lock is granted. The transaction holding the DELAY lock is entered on the DELAYing transaction list for the requesting transaction. The requesting transaction does not block, and hence a process switch is avoided.

If the transaction holding the READ lock is ACTIVE (not PREPAREd), then a write request is treated as a read-write conflict in which the requesting transaction must block. The transaction holding the READ lock is entered on the DELAYing transaction list for the requesting transaction in anticipation of the downgrading of these locks.

When a transaction holding READ locks PREPAREs, it downgrades its READ locks to DELAY locks. This is accomplished by unblocking all transactions that had requested WRITE locks on its READ locked data while it was ACTIVE. These blocked WRITE-requesting transactions need to be identified so that they can be permitted to proceed. This is the only burden placed on the holders of DELAY locks. Transactions without blocked writers do not pay this cost.

When the WRITE-requesting transaction PREPAREs, its time range vote must be cast. The DELAYing list is scanned. Terminated transactions on the DELAYing list are ignored. If all transactions are terminated (either COMMITted or ABORTed), then the time range vote is unaffected by DELAY locks. Otherwise, the latest LATEST vote of all the still PREPAREd transactions on the DELAYing list becomes the lower bound on the EARLIEST vote for this transaction.

4.4 Two Phase Locking and Two Phase Commit

It is easy to overlook a fundamental assumption in much of the discussion of two phase commit and its optimizations. This assumption is that all non-commit related processing in all cohorts of a transaction has terminated prior to the commit protocol beginning. In particular, no activity requiring the locking of additional data is continuing. This assumption is straightforward to guarantee when all processing follows the request/response paradigm. The coordinator only initiates the 2PC protocol when all responses have been received.

Not all systems require the request/response paradigm. And for these, assuring that locking of data has terminated will typically require extra messages. In the absence of this condition, any cohort's release of READ locks at PREPARE time may violate two phase locking.

4.4.1 Example:

```
Cohort C1 of a transaction releases read locks when the PREPARE message arrives.
Cohort C2 receives the PREPARE message somewhat later, and continues to acquire
locks during this period.  Hence the locking for the entire distributed
transaction is not two phased, even though it is two phased at each cohort.
A second transaction may then be able to change C1's released data, hence
serializing after C1,  and also change data prior to C2 examining it, hence
serializing before C2.  Thus, the global transactions are not serializable.
```

In this case then, no optimization that releases read locks at prepare time can be permitted, because serialization cannot be guaranteed. This precludes the read-only optimization.

Now, however, consider the timestamping 2PC protocol. Each database system is locally two phased with respect to lock acquisition. This local two phased property, together with DELAY locks, ensures that locally conflicting transactions have disjoint PREPAREd-INTERVALs. Hence, local transactions will have transaction timestamps ordered correctly locally. And globally, the commit protocol ensures that the timestamp order of the transactions correctly orders transactions. Thus, even in the absence of global two phase locking, the timestamp order chosen agrees with ALL local serializations.

Essentially, two phase locking is being used locally, up to PREPARE time, to order transactions. Then timestamp order concurrency control is used. This offers high concurrency with the efficiency of using the commit protocol itself to "quiesce" the transaction cohorts' normal activities, without a separate termination protocol. In particular, it makes it possible to exploit the 2PC protocol messages to trigger "delayed" constraint evaluation, while still assuring serializability of transactions.

Some care must be taken here. As more activity is permitted to follow the initiation of the commit protocol, more time must be allowed for cohorts to complete their diverse activities. If timestamp ranges are not sufficiently large, the probability that their intersection is empty, forcing transaction abort, increases.

5 Dealing with Non-Timestamping Cohorts

5.1 A Problem with Heterogenous Systems

In a heterogenous system, not all cohorts of a transaction necessarily timestamp their data. We would like our commit protocol to work correctly when transactions involve both timestamping and non-timestamping database cohorts. If the non-timestamping cohort does not include a timestamp on its voting message, then a problem arises. Even though transactions are serialized correctly at each database, and a valid global serialization for all databases is assured, the timestamp order cannot be guaranteed to agree with a valid global serialization.

5.1.1 Example:

```
Transaction T1 executes at timestamping database A and non-timestamping
database B.  Transaction T2  executes at non-timestamping database B and
at timestamping database C.  Transaction T1 commits at B prior to T2.
However, the EARLIEST time voted for T1  at A is later than the EARLIEST
time for T2 at C.  Since there are no constraints established at B, these
times can become the transaction times.  They satisfy the local constraints
at A and C, but they do not agree with a valid serialization of T1 and T2,
which  must have TI ordered before T2.
```

5.2 The Role of the Transaction Manager

It is useful to introduce the notion of a system component called the transaction manager (TM). The TM exists at every node in the system and assists the database systems on each node to coordinate distributed transactions. It does this by presenting a strictly local interface to each database system through which the two phase commit protocol is exercised. The TM performs the communication required in the commit protocol. That is, any commit protocol message has a source that is a TM at one site, and a destination that is a TM at another site.

A node's TM interfaces with all databases at the node, whether timestamping or non-timestamping. The TM coordinates the transaction, at the direction of one of its local databases. Since the TM exists at every node, any node can coordinate the transaction, whether or not a timestamping database is present. Each database system notifies its local TM about commit initiation and voting. The coordinator TM examines votes, decides whether to commit or abort a transaction, and selects the transaction time. It then communicates to other remote participating TMs the transaction dispostion and time. These TMs inform their local participating databases.

5.3 Transaction Manager Voting

The solution to the problem of mixed timestamping and non-timestamping databases in the same transaction is for the TM to provide a timestamp should a database not inform the TM of an EARLIEST time. The TM executes the procedure in 2.2.1 to choose an EARLIEST timestamp. It keeps the LAST variable for **each** database system with which it deals on the node.

Note here that a TM interacting with a database on its node can also supply the LATEST, i.e. high bound, for the transaction time vote should the database itself not provide it. This is similar to the TM role when dealing with a non-timestamping database. But now, the TM can supply either EARLIEST, LATEST, or both bounds. All of these alternatives are potentially useful.

With a TM, a database system need not know anything about timestamps. And the TM need know very little about the database. The TM executes the timestamp selection protocol in the absence of a transaction time vote. The TM can execute only the first alternative of step 2 of procedure of 2.2.1 to choose an EARLIEST time. A timestamping database system might be able to vote an earlier EARLIEST time. We assume that the TM does not have access to the more detailed information needed for an earlier vote.

What enables timestamp ranges to ensure transaction serialization is the enforcement by each database of disjoint PREPAREd-INTERVALs for conflicting transactions. A database usually does this via strict two phased locking. If a database communicating with a TM is known to to guarantee this, then not only is serializability ensured, but all of the previous optimizations of the timestamping databases are possible. A non-timestamping database might even employ DELAY locks, holding them until a transaction has committed, in its role of enforcing the guarantee.

5.4 Disjoint PREPAREd-INTERVALs for ALL Transactions

In a heterogenous system, the TM cannot depend on all databases ensuring disjoint PREPAREd-INTERVALs. For example, if a non-timestamping database releases read locks at PREPARE time, and does not use DELAY locks, then conflicting transactions might be simultaneously PREPAREd. This does not compromize serializability, assuming that all locking is completed prior to the commit protocol initiation [see section 4.4]. However, it can cause the timestamp order to differ from a valid serialization.

If the TM has no information about a local database's behavior in this regard, then the TM must ensure disjoint PREPAREd-INTERVALs for conflicting transactions by itself. One idea is to prevent ANY transactions, not merely conflicting ones, from being simultaneously PREPAREd. This clearly keeps conflicting transactions from being simultaneously prepared. The TM can realize this very simply by requiring one transaction from the database to commit before the next transaction is prepared.

A variation of this approach enforces this constraint by exploiting timestamp ranges. The TM can ensure disjoint PREPARE-dINTERVALs by how it votes timestamp ranges. When the TM votes an [EARLIEST, LATEST] timestamp range for a transaction, the EARLIEST time must be later than not only the LAST commit time but also the latest LATEST upper bound voted by all currently PREPAREd transactions. Thus,

$$EARLIEST_i(X) = \max\{LAST_i(X) + \epsilon, CLOCK_i(X), \max\{LATEST_i(Y) | PREPAREd_i(Y, X)\}\}$$

5.5 Preventing Early Lock Release

The above demonstrates that an appropriately designed "timestamping" TM can cope with database systems that expect to use ordinary 2PC and to release READ locks at prepare time. However, the designs can seriously impact performance. The problem is that transactions are essentially "single-threaded" through the PREPAREd state. A heavily used database system will experience this as a bottleneck to high performance. For such database systems, the best way of limiting the enforcement of disjoint PREPAREd-INTERVALs to conflicting transactions may well be to retain all locks until commit and to give up the early lock release optimizations.

If we know that a database system uses two phase locking, with no release of locks prior to PREPARE, the TM may be able to prevent the database system from releasing locks until commit time. If the database system waits for an ACK to its PREPARE vote before releasing locks at PREPARE time, then the TM can delay the ACK for message two until commit time. If the database uses two phase locking up to prepare time, then this two phase locking becomes strict two phase locking when combined with the delayed ACK. This guarantees that PREPAREd-INTERVALs of conflicting transactions are disjoint. Hence, timestamp order will agree with serialization order.

6 Discussion

Stamping data with the transaction time of the updating transaction permits database systems to support multiple versions and to answer queries about the state of the database AS OF some time in the past. It is desirable to choose the transaction time late in the transaction so as to maximize concurrency. We exploit an enhanced 2PC protocol, using the same number of messages as the normal 2PC protocol, as a mechanism for reaching agreement among cohorts as to what the transaction time should be.

Our enhanced 2PC protocol permits us to exploit the common optimizations that normally can be used with ordinary 2PC. In particular, we showed how a transaction's read locks can be released at PREPARE time. Importantly, our enhanced 2PC can exploit this even when normal transaction activity cannot be guaranteed to be complete. This is not supported by the ordinary 2PC protocol.

Finally, we showed how our protocol could work in a heterogenous system containing non-timestamping databases. This permits the timestamping databases to interoperate with non-timestamping ones while continuing to assure that the timestamp order of transactions agrees with a valid serialization order for all global transactions.

7 Acknowledgements

Discussions with Phil Bernstein, Jim Johnson, and Ken Wilner aided in the development of the timestamping 2PC protocol. Phil Bernstein and Betty Salzberg provided useful comments on an earlier draft of this paper.

8 Bibliography

[BER] Bernstein, P., Hadzilacos, V. and Goodman, N. Concurrency Control and Recovery in Database Systems. Addison-Wesley Publishing Co., Reading MA (1987).

[GRA] Gray, J. Notes on Database Operating Systems. in "Operating Systems: an Advanced Course", Lecture Notes in computer Science 60, Springer-Verlag, (1978) 393-481.(also IBM Research Report RJ2188, Feb. 1978)

[HER] Herlihy, M. Optimistic Concurrency Control for Abstract Data Types. Proceedings of Principles of Distributed Computing Conference (1986) 206-217.

[JOS] Joshi, A. and Rodwell, K. A Relational Database Management System for Production Applications. Digital Technical Journal 8 (Feb. 1989) 99-109.

[LAMO] Lamport, L. Time, Clocks, and the Ordering of Events in a Distributed System. Comm ACM 21,7 (July 1978) 558-565.

[LAMS] Lampson, B. and Sturgis, H. Crash Recovery in a Distributed System. Xerox PARC Research Report, 1976.

[LOM] Lomet, D. and Salzberg, B. Access Methods for Multiversion Data. Proceedings of the ACM SIGMOD Conference, Portland, OR. (May 1989), 315-324.

[REE] Reed, D. Implementing Atomic Actions. Proceedings of the 7th Symposium on Operating System Principles (1979) and in ACM Transactions on Computer Systems, 1,1 (Feb. 1983) 3-23.

[SNO] Snodgrass, R. and Ahn, I. A Taxonomy of Time in Databases. Proceedings of the ACM SIGMOD Conference, Austin, TX (May 1985), 236-246.

[STO] Stonebraker, M., The Design of the POSTGRES Storage System. Proceedings of the 13th VLDB Conference, Brighton, UK (Sept. 1987), 289-300.

Table 1: Definitions of Terms

Terms	Definitions
Time Terms	
$CLOCK_i(X)$	clock time at site i when transaction X prepares
$EARLIEST_i(X)$	lower bound for time that is acceptable to site i for transaction X
$LAST_i(X)$	time of last committed transaction at site i when transaction X prepares
$LATEST_i(X)$	upper bound for time that is acceptable to site i for transaction X
$TIME(X)$	transaction time for transaction X
Predicates	
$COHORT_i(X)$	does site i have a cohort of transaction X
$CONFLICTS_i(Y, X)$	does transaction Y in PREPAREd state at site i conflict with transaction X at site i when transaction X prepares
$PREPAREd_i(Y, X)$	is transaction Y in PREPAREd state at site i when transaction X prepares