Using Timestamping to Optimize Two Phase Commit

David Lomet
DEC Cambridge Research Lab
One Kendall Sq., Cambridge, MA 02139

Abstract

The two phase commit (2PC) protocol is used to guarantee the serializability of distributed transactions. The message cost of the standard 2PC has led to efforts to optimize the protocol and reduce the number of messages required. The common optimizations require that each cohort of a transaction be terminated (finished with normal accessing of data) in order for these optimizations to lead to serial schedules. This paper suggests using timestamps as a substitute for knowing when cohorts are terminated, and shows how the 2PC protocol itself can be used to choose the timestamps. The key to this is to permit cohorts to vote transaction time ranges within which the transaction must commit or else be aborted. Using time ranges, the read only optimization and early release of read locks can be supported. The transaction times chosen are appropriate for identifying versions of data in a multiversion rollback database.

1 Introduction

1.1 Two Phase Commit

Two phase commit [2, 6] is a protocol that is used to ensure the serializability of distributed transactions. This protocol has two phases, as indicated in its name. The end of a transaction's active phase, where it is executing "normally", is signalled by a PREPARE message, while the transaction's final disposition is indicated by a COMMIT or ABORT message, usually after the transaction has "prepared". Each of these messages is routinely acknowledged. Hence, the usual message cost of two phase commit(2PC) is four messages per transaction participant (cohort).

Optimizations to the 2PC protocol have been designed to reduce the above message cost. Most optimizations [9] rely on the assumption 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. Ensuring termination is what causes a major part of the complexity of commit protocol implementations. In the absence of ensured termination, these optimizations do not guarantee serializability.

1.2 Cohort Termination

1.2.1 The Problem

Termination 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. However, not all systems follow this paradigm. And for those that don't, ensuring that locking has terminated can require extra messages. Without some further effort, termination is only known to cohorts when they receive the COMMIT/ABORT message.

No optimization that releases locks prior to termination, e.g., at prepare time, can be permitted because serialization cannot be guaranteed. This precludes the read-only optimization, where a cohort votes "Read Only' at phase one of the protocol and does not participate at phase two. Such a cohort never finds out when all cohorts of a transaction are guaranteed to have terminated. Releasing its locks risks non-serializability. Below is an example where the early release of READ locks during the commit protocol violates two phase locking, and hence compromises serializability.

Example:

Cohort C1 of a transaction releases read locks when the 2PC 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 transaction is not two phased, even though it is two phased at each cohort. A second transaction may 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.

1.2.2 Some Examples of the Problem

Most commercial transaction management protocols support non-request/response functionality, e.g. IBM's LU6.2 and Digital's DECdtm [1]. They do this because the request/response paradigm is too restrictive in several important cases. We describe two examples below.

Constraint Evaluation: Even assuming that a database system normally uses request/response for database operations, e.g. requesting data from remote systems, there is a performance advantage to not requiring separate requests to trigger delayed constraint evaluation. These requests might well have to reach all cohorts, exactly like the PREPARE message itself. It is more efficient to

[1] LU6.2 is a trademark of the IBM Corporation, DECdtm is a trademark of Digital Equipment Corp.

use the PREPARE message to trigger the constraint execution. But now the transaction may not be terminated when the commit protocol is initiated. New read locking may well be required to evaluate the constraint.

Data Streams: Request/response is cumbersome if one's intent is to initiate data streams flowing between transaction cohorts, say to perform distributed joins in a pipelined fashion. It is also the case that a user may not want to complete all requests that are streaming him answers before wrapping up the transaction and committing. That is, he may have done some updating, looked at some partial answers, and then want to commit the transaction. But interrupting such stream processing early is not an instance of request/response. Here again, a transaction is being terminated while work is going on.

In both the cases above, transaction termination can only be ensured prior to commit protocol initiation if an additional round of messages were to travel to all possibly active cohorts. This would turn 2PC into 3PC, where the first phase is performed to ensure termination. This surely obviates most of the benefit from these optimizations. Our timestamping approach permits the interesting optimizations without requiring ensured termination, and hence without needing an additional phase.

1.2.3 Coping Using Timestamps

Our purpose here is to demonstrate how **timestamps** that correctly serialize transaction can overcome the difficulties introduced by the uncertainty of cohort termination. In particular, 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 time. This is done without extra message overhead, as suggested in [4, 11].

We suggest here that cohorts vote on the transaction time using bounded time ranges. It is the bound on the range of time during which a transaction must commit, and the timestamp order that is enforced by the protocol, that compensate for the lack of information concerning the termination of cohorts. A cohort can be sure that all other cohorts to a transaction have terminated after the time given for the transaction. Hence, after the transaction time, no further locking is going on at any cohort, and locks at all cohorts can be released. This transaction time can, when time ranges are voted, be bounded, permitting locks to be freed without the need to receive the commit message.

1.3 Organization of the Paper

We show how timestamps or timestamp ranges can be decided upon and voted by each cohort and how timestamps can be agreed upon and propagated using the two phase commit protocol in section 2. This permits us to exploit commit protocol optimizations which ensure that distributed transactions are correctly serialized even when their locks are not strict two phased. This is discussed in section 3. Section 4 discusses how to interoperate in a heterogeneous system where not all database systems perform timestamp voting. Section 5 reviews the impact of timestamps on the two phase commit protocol and discusses the further uses of our timestamping two phase commit in multiversion databases.

2 Agreeing on Transaction Time

2.1 The Two Phase Commit Protocol

We begin with an informal description of the two phase commit (2PC) protocol for coordinating distributed transaction. It has the following steps:

1. A coordinator notifies all transaction cohorts that the transaction is to be terminated, via the PREPARE message (message one of the protocol).

2. Each cohort then sends a VOTE message (message two) on the disposition of the transaction. The vote is either COMMIT or ABORT. A cohort that has voted COMMIT is now PREPAREd.

3. 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 disposition message (i.e. COMMIT or ABORT) (message three) to all cohorts.

4. The cohort terminates the transaction according to its disposition, either COMMIT'd or ABORT'd. The cohort then ACKs(message four) the disposition message.

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 3 and describe extensions that work with these.

2.2 Voting for Transaction Time

2.2.1 The Basic Protocol

To select a transaction time, we extend the 2PC protocol by augmenting the information conveyed on two of its messages. 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 disposition is COMMIT, the transaction time chosen.

2.2.2 Cohort Time Selection

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. 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 [5] to ensure that transaction times are monotonically increasing.

A site (database system at the site) that executes the following procedure will generate a time for a transaction at the site that is later than the transaction times of all previously committed transactions with which the committing transaction conflicts. (Note: All notation used in equations here and subsequently are defined in Table 1.)

1. A database system maintains a monotonically increasing $LAST_i$ transaction time at each site i. It does this by comparing $LAST_i$ 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 $LAST_i$, $LAST_i$ is set to the value of this new timestamp. This is the Lamport clock component. $LAST_i$ is also advanced whenever a local transaction chooses a transaction timestamp and commits. Thus, $LAST_i(X)$, the value of $LAST_i$ at the time transaction X commits, is an upper bound on the timestamps of earlier conflicting transactions.

   A more aggressive alternative is to save $LAST_i$ as $LAST_i(X)$ whenever X acquires a lock. The final $LAST_i$ so saved is surely later than all conflicting earlier transactions when locks are held until commit time. (Note: we cannot choose an earlier time than this without knowing the transaction times of transactions that have read the data that this transaction writes. But this risks turning data reads into writes so that we can record this information. See section 3.5 below for another approach to this.)

   Thus our initial definition of $CONFLICT_i(X)$, a time guaranteed to be later than the time of all earlier conflicting transactions, is

   $$CONFLICT_i(X) = \epsilon + LAST_i(X)$$

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 both the time of any preceding conflicting transaction in that database. So, 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 no earlier than $CONFLICT_i(X)$. Further, a transaction's commit time should come no earlier than its start time, $START(X)$, to keep the transaction time synchronized with user expectations. Thus, cohort i votes a time for transaction X of

   $$EARLIEST_i(X) = \max(CONFLICT_i(X), START(X))$$

3. The coordinator picks a transaction time that is not earlier than the latest $EARLIEST$ time voted by 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 earliest such time. This 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. 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

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

2.2.3 Timestamp and Serialization Order

We call the time between a cohort's $EARLIEST$ vote and the commit time of the transaction the PREPAREd INTERVAL. The result of item 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.

$CONFLICT_i(X)$ is later than the time of any previously committed conflicting transaction. A following transaction at a site will thus vote an $EARLIEST$ time that is later than earlier conflicting transactions. The agreed upon time will thus be later than all conflicting transactions at all sites. This ensures 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.

2.3 Timestamp Ranges

2.3.1 Divergent Clock Time

A difficulty for our timestamping 2PC extension is that a database cohort with a substantially faster clock can seriously disrupt the entire distributed system and the transaction times that are chosen. A late $EARLIEST$ vote will always become the transaction time. This forces transaction time ahead of clock time at cohorts whose clocks are running correctly. This is bad because if a cohort commits 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 with $EARLIEST$ votes 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 mechanism of dealing with this.

2.3.2 Voting With Timestamp Ranges

One way to establish bounds that limit the divergence of $EARLIEST$ votes is to have the cohorts 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

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

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.

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.

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 during which transactions are in doubt. 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.

3 Optimizations of the 2PC Protocol

3.1 The Read-Only Optimization

When transaction termination is guaranteed prior to the initiation of the 2PC protocol, a read-only cohort, i.e., one that has no updates, does not need to receive the COMMIT message as it has no activity that it needs to perform as a result. It merely releases its locks when it receives the PREPARE message. With timestamps, 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 ensures that timestamp order agrees with serialization order.

3.2 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's data may be unavailable for 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 and transaction termination is guaranteed prior to initiation of the commit protocol. Again, the constraint that timestamping requires, i.e. that two conflicting transactions not be simultaneously prepared, limits our response to blocked transactions. That is, we must ensure that subsequent transactions that commit after this prepared transaction have later timestamps. But, having done this, we can then exploit this optimization, even when transactions have not terminated prior to commit protocol initiation.

With cohorts voting a 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.

3.3 Releasing Read Locks at PREPARE

Without timestamping requirements, any cohort can release READ locks at PREPARE time, when 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 this prior prepared transaction. The general problem here is to keep PREPAREd INTERVALs disjoint for conflicting transactions so that PREPAREd order becomes COMMIT'd order and timestamp order as well. Hence, this problem is one of ensuring that a subsequent conflicting transaction votes an $EARLIEST$ time that is later than the $LATEST$ time that is voted by the current transaction.

One approach is to force $LAST$ to immediately be set to the $LATEST$ time voted. This is unlikely to be satisfactory, however. It increases the divergence between clock time and transaction time, which may lead to unnecessary transaction abort or to user surprise concerning transaction time.

3.4 Delaying Conflicting Transactions

3.4.1 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. At PREPARE time, a transaction transforms all its read locks to DELAY locks. At commit time, the DELAY locks are 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 associate a time with DELAY locks. This time is examined should DELAY locks still be 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. To be sure a transaction gets a timestamp later than all conflicting transactions, we redefine $CONFLICT_i(X)$ as

$$CONFLICT_i(X) = \epsilon +$$
$$\max(\{LAST_i(X)\} \cup \{LATEST_i(Y) \mid DELAYS_i(Y, X)\})$$

This definition of $CONFLICT_i(X)$ ensures that conflicting transactions continue to have disjoint PREPAREd INTERVALs, and hence that timestamp order and serialization order agree.

3.4.2 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 PREPAREs, it demotes 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. Completed (COMMIT'd or ABORT'd) transactions on the DELAYing list are ignored. If all transactions are completed, 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.

3.5 Timestamped Locks

An earlier $EARLIEST$ vote increases the size of the timestamp range that can be voted, hence increasing the probability that a transaction can be committed, without increasing the time of the $LATEST$ vote. The preceding has shown how timestamps can be used to realize delay locks by permitting early release of resources while delaying the timestamp of subsequent conflicting transactions. We can exploit timestamps on locks to also permit transactions to vote an earlier $EARLIEST$ time.

Normally, the lock manager purges locks held by committed transactions when they commit. This means that the system no longer knows the last time at which data has been read by committed transactions. However, when we convert read locks to DELAY locks instead of releasing them completely, we retain that information. DELAY locks have been released when their holding transactions committed. But to permit read-write conflict detection, we can choose to continue to hold DELAY locks for a more extensive period. Thus, the system would know the timestamps of recently read data and subsequent writers would be able to define $CONFLICT_i(X)$ more precisely.

What the above entails is to include in the definition of $CONFLICT_i(X)$ explicit information about the set of recent conflicting transactions, i.e. those that must serialize $BEFORE$ the committing transaction. When pursuing this course, write locks can also be timestamped when a transaction commits, permitting us to detect write-write and write-read conflicts as well as read-write conflicts. Instead of purging locks when a transaction commits, we continue to maintain them in the lock manager for some specified time period Δ, perhaps some modest number of seconds, as if they were DELAY locks. (We can deal with write-read and write-write conflicts in a timestamping multiversion database (see section 5.2) by timestamping the changed data.)

When a subsequent transaction X requests a lock that conflicts with one of these timestamped locks, the new lock is granted, but, similar to DELAY locks, the transaction holding the lock is recorded in association with the requesting transaction and the requesting transaction is required to commit after this earlier conflicting transaction. Since only transactions conflicting in the last Δ seconds are included in $BEFORE_i(Y, X, \Delta)$, one cannot be sure of conflicts more than Δ seconds ago. Thus

$$\begin{aligned} CONFLICT_i(X) = \epsilon + \\ \max(\{TIME(Y) \mid BEFORE_i(Y, X, \Delta)\} \\ \cup\{LATEST_i(Y) \mid DELAYS_i(Y, X)\} \\ \cup\{\min(CLOCK_i(X) - \Delta, LAST_i(X))\}) \end{aligned}$$

This last definition of $CONFLICT_i(X)$ is the most precise information we have on the time of the latest conflicting transaction, and can be used instead of earlier definitions when determining the $EARLIEST_i(X)$ vote.

4 Votes Without Timestamps

4.1 Heterogeneous Systems Problem

We would like our commit protocol to work correctly in a heterogeneous system where transactions may involve both timestamping and non-timestamping database cohorts. If a 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 ensured, the timestamp order cannot be guaranteed to agree with a valid global serialization.

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. Let transaction T1 commit at B prior to T2. However, the EARLIEST time voted for T1 at A can be later than the EARLIEST time for T2 at C since there are no constraints established at B. Hence, the timestamps for T1 and T2 do not necessarily agree with a valid serialization of T1 and T2, which must order T1 before T2.

4.2 The Role of the Transaction Manager

Distributed transaction processing systems have a system component called the transaction manager(TM) [3]. The TM exists at every site in the system and assists the database systems at each site 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 site's TM interfaces with all databases at the site, whether timestamping or non-timestamping. The TM coordinates the transaction, at the direction of one of its local databases or application programs. Since the TM exists at every site, any site can be a transaction coordinator, 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 disposition and time. These TMs inform their local participating databases.

4.3 Transaction Manager Voting

The solution to the problem of non-timestamping databases in a transaction is for the TM to provide a timestamp should a database not vote an $EARLIEST$ time. The TM executes the procedure in section 2.2.2 to choose an $EARLIEST$ timestamp. It keeps the $LAST$ variable for **each** database system with which it deals at the site.

Note that a TM interacting with a database on its site 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. These alternatives are all 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 an $EARLIEST$ vote. The TM can execute only the first alternative of 2.2.2 to determine $CONFLICT_i(X)$. A timestamping database system might be able to do better (see, e.g., section 3.5). We assume that the TM does not have access to the more detailed information needed for this.

What enables timestamp ranges to ensure transaction serialization is that each database enforces disjoint PREPAREd INTERVALs for conflicting transactions. Database usually do this via strict two phased locking but it might employ DELAY locks or timestamped locks in its role of enforcing disjoint PREPAREd INTERVALs. If a database communicating with a TM is known to guarantee this, then not only is serializability ensured, but all of the previous optimizations of the timestamping databases are possible.

4.4 Disjoint PREPAREd INTERVALs

The TM may not be able to 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 compromise serializability, assuming that all locking is completed prior to the commit protocol initiation. 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 itself must ensure disjoint PREPAREd INTERVALs for conflicting transactions. One idea is to prevent ANY transactions, not merely conflicting ones, from being simultaneously PREPAREd, which is clearly sufficient. 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 that conflicting transactions have disjoint PREPAREd INTERVALs by using a conservative definition of $CONFLICT_i(X)$ which exploits the knowledge that it has about the $LATEST$ votes of ALL prepared transactions. Thus

$$CONFLICT_i(X) = \epsilon +$$
$$\max(\{LAST_i(X)\} \cup \{LATEST_i(Y) \mid PREPARED_i(Y, X)\})$$

4.5 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. The two phase locking then becomes strict two phase locking when combined with the delayed ACK. This guarantees that PREPAREd INTERVALs of conflicting transactions are disjoint.

4.6 Registering with the TM

There are substantial differences in performance for each of the above database types in their interaction with the TM. This suggests that a TM be designed to recognize the database type and so provide it with the best possible performance permitted for the type. Typically, databases register with the TM at a site so as to make known their 2PC protocol entry points to the TM. This registration process could also serve to identify to the TM the type of database.

We expect database systems to have only a transitional period in which they use the single threaded commit protocol. Competitive pressure should force their evolution to higher performance.

5 Discussion

5.1 Timestamping Two Phase Commit

A database system may not want cohorts terminated prior to 2PC protocol initiation. For example, a database can use the PREPARE message to invoke "delayed" constraint evaluation or delayed triggers. Our timestamping 2PC protocol permits this while still ensuring serializability of transactions. Further, it can exploit common 2PC optimizations that normally can be used only when termination is guaranteed. This offers high concurrency with the efficiency of using the commit protocol to "quiesce" transaction cohorts. With an appropriate transaction manager, our protocol works in a heterogeneous system containing non-timestamping cohorts

Choosing sizes for timestamp ranges voted by cohorts requires making trade-offs between concurrency and the frequency of aborts. The larger the range, the more likely that transactions commit, but the less concurrency. This is not unlike other choices for timeouts, e.g. for deadlocks or in network communication protocols. The substantial payoff for voting timestamp ranges is the elimination of termination messages and the potential for early release of locks.

5.2 Multiversion Databases

5.2.1 Transaction Time Databases

Transaction time databases [8] use transaction time to stamp each version of data in a multiversion database.

The order of the timestamps must be a correct serialization of the transactions. Time-slice queries retrieve data a transaction consistent view of the database as of some past time. Data that is current may continue to be updated, and hence is best stored on a medium that can be multiply written, e.g. magnetic disk. However, "historical" data is never updated, and hence could be stored on write-once, read many (WORM) optical disks [10, 7]. Inexpensive WORM optical disks change dramatically the functionality/cost trade-off and may make transaction time databases useful for a large number of applications.

5.2.2 Providing Transaction Timestamps

Timestamping methods impose serialization when the timestamp is chosen [1]. Competing requests from transactions with timestamps ordered differently from request order require that one of the transactions be aborted. When a timestamp is chosen at transaction start, it is known to all cohorts during updating, and can be used to stamp the data then. However, dealing with read-write conflicts can require that data items carry the timestamp of the last reader, changing reads into writes.

Two phase locking remains the concurrency method of choice. Locking has understood and acceptable performance. Locks delay conflicting requests without reference to timestamps. Locking permits us to choose timestamps at commit time that correctly reflect the serialization that transactions actually experienced. The timestamping 2PC protocol facilitates this by avoiding extra messages for time agreement.

With transaction time unknown until commit, it is impossible to post timestamps during updates. What is needed is either a second visit to the updated data, or a persistent way of associating transaction time with a transaction identifier stored with the data. Combinations of these strategies are also possible so as to enable lazy posting of the timestamps (see [10, 8]).

Acknowledgments

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

References

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

[2] 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 in IBM Research Report RJ2188 (Feb. 1978).

[3] Gray, J. and Reuter, A. *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann Publishers, Inc., San Mateo CA (in preparation).

[4] Herlihy, M. Optimistic Concurrency Control for Abstract Data Types. *Proc. Symp. on Principles of Distributed Computing*, (1986) 206-217.

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

[6] Lampson, B. and Sturgis, H. Crash Recovery in a Distributed System. Tech Report (1976) Xerox PARC, Palo Alto CA.

[7] Lomet, D. and Salzberg, B. Access methods for multiversion data. *Proc. ACM SIGMOD Conf.*, Portland, OR (May 1989) 315-324.

[8] Lomet, D. and Salzberg, B. Rollback Databases. Digital Equipment Corp. Tech Report CRL92/1 (Jan 1992) Cambridge Research Lab, Cambridge, MA..

[9] Mohan, C., Lindsay, B. Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactions. *Proc. Symp. on Principles of Distributed Computing*, Montreal, CA (Aug. 1983)

[10] Stonebraker, M. The Design of the POSTGRES Storage System. *Proc. Very Large Databases Conf.*, Brighton, UK (Sept. 1987), 289-300.

[11] Weihl, W. Distributed Version Management for Read-Only Actions. *IEEE Trans. on Software Engineering* SE-13,1 (Jan. 1987), 55-64.

Table 1: Definitions of Terms

Terms	Definitions
Symbols	
X, Y	distributed transactions
i	site of a distributed system
T	current time
Time Terms	
$CLOCK_i(X)$	T at i when X prepares
$EARLIEST_i(X)$	earliest time acceptable to i for X
$LAST_i$	time of last committed transaction at i
$LAST_i(X)$	$LAST_i$ when X prepares or last locks
$CONFLICT_i(X)$	time later than conflicting transactions at i for X
$LATEST_i(X)$	latest time acceptable to i for X
$START(X)$	start time for X
$TIME(X)$	transaction time for X
Predicates	
$COHORT_i(X)$	does site i have a cohort of X
$DELAYS_i(Y, X)$	does Y in PREPAREd state at i delay X at i when X prepares
$PREPARED_i(Y, X)$	is Y in PREPAREd state at i when X prepares
$BEFORE_i(Y, X, \Delta)$	must Y, with $TIME(Y) > T - \Delta$, at i serialize before X