USING HISTORY INFORMATION TO PROCESS DELAYED DATABASE UPDATES

Sunil K. Sarin, Charles W. Kaufman, and Janet E. Somers

Computer Corporation of America
Four Cambridge Center
Cambridge, Massachusetts 02142

Abstract: An algorithm is described which processes database updates arriving out of order in a way that maintains a consistent view of the data. This problem arises in the context of a high availability replicated database architecture in which updates are totally ordered by timestamp but do not necessarily arrive at a site in timestamp order. The algorithm uses a history of object values written and objects read by updates. When a new update arrives and is executed, higher-timestamped updates that read its results are scheduled for undoing and reexecution; such reexecution may in turn cause additional updates to be reexecuted, and so on. A major goal of the algorithm is to avoid this kind of cascading when reexecution of an update would have the same effect as it had before. A prototype implementation of the algorithm for a relational database is described. It is suggested that the algorithm may be of use outside its original context, in the maintenance of historical databases.

1. Introduction

Replication of data at multiple sites offers the potential for high availability of a database in a distributed environment. However, the need to maintain consistency of multiple copies usually limits the availability that can be achieved in the face of communication failures. If the communication network is partitioned into two or more disconnected groups, at most one group (the one, if any, with a majority of copies or votes [16,8]) will be allowed to update a given data item; transactions in other groups cannot even read the data item and update others without incurring the risk of violating serializability. This may be unacceptable in applications — such as inventory control, banking, and reservation systems — that require continued service in the presence of communication failures and partitions.

The System for Highly Available Replicated Data (SHARD) developed at CCA emphasizes continued operation of a replicated database in spite of network partitions and other communication failures [15]. Transactions in SHARD do not execute serializably. Database queries and external actions issued by a transaction are performed immediately at the site executing the transaction. However, updates issued by a transaction are processed asynchronously by the sites in the system, after some communication delay which is highly variable (especially if there is a network partition) and which may be different for each site.

Because transactions are not serializable, integrity constraints on the database may be violated. Applications that use SHARD must distinguish between structural "invariants" that are required to always be true for the stored data (e.g., computed totals consistent with base data) and more general "semantic integrity constraints" that represent desirable states but need not be strictly enforced (e.g., no overbooking of seats or overdrawing of money). The update parts of transactions are designed to preserve the invariants, but integrity constraints may be violated and inconsistent external output may be issued. We rely on the application to compensate for these problems when they occur. It is also possible for the application to selectively introduce "pessimistic" concurrency control, giving up some availability, to reduce the likelihood of inconsistency and cost of compensation.

This paper is not concerned with how applications deal with non-serializable transaction execution (described in [14]), but with the processing of updates in a way that preserves mutual consistency of multiple database copies and supports the application's compensation requirements. Updates issued by transactions are assigned unique *timestamps* [11], and a site's database copy at any given time must reflect those updates that it has seen so far as if they had been executed in timestamp order. Because of unpredictable communication delays, an update arriving at a site may have lower timestamp than some

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and by the Air Force Systems Command at Rome Air Development Center under Contract No. F30602-84-C-0112. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of DARPA, RADC, or the U.S. Government.

previously-received and already-executed updates. If any of these higher-timestamped updates conflict with the newly-received one, they must be undone and reexecuted to maintain the desired timestamp ordering criterion. The objective of the algorithm we present is to control the amount of such reexecution that is needed, while expending a moderate computational effort in determining which updates to reexecute.

Previous proposals to use timestamp ordering for mutual consistency assumed that every update unconditionally *overwrites* a specified data item with a given new value. Mutual consistency is achieved by the following well-known method, first proposed by Johnson and Thomas [10]. The timestamp of the highest-timestamped update to each data item is remembered, and a newly-arriving update to a given data item is ignored if its timestamp is smaller than the timestamp currently associated with the data item. Updates that add or delete items to and from a set (as in [2,5]) are equivalent to overwriting the "membership function" of items in the set to true or false, and can be treated similarly.

Updates in SHARD can be considerably more complex than simple overwrites. The reasons for this are twofold. First, we wish to permit specialized update operations, such as incrementing and decrementing a numeric quantity. Consider the following transaction that withdraws money from a bank account whose balance is stored under the data item "Balance":

```
amount := ask-user("How much to withdraw?")
if read(Balance)>=amount then
    give customer amount in cash
    issue update: Decrement(Balance,amount)
```

The Decrement operation above takes effect at a given site at the time of execution of the update, which may be some time later than the time of reading the balance and dispensing the money. If two concurrent withdrawals were to see the same initial balance, the final value of the balance (after both Decrement operations have been received and executed) would correctly reflect the total amount withdrawn.

The second reason for including more complex update types is to allow the detection of nonserializable execution so that the application can perform compensating actions. For example, if the initial balance of an account were $400, two concurrent withdrawals of $200 and $300 would both succeed (because each reads a balance that appears to be sufficient) but would leave a negative final balance, −$100. The bank may wish to compensate for this whenever it does occur, perhaps by assessing a fine and sending a letter to the customer demanding payment. To support this, the withdrawal interaction should issue a more complex update of the following form:

```
Decrement (Balance,amount)
if Balance<0 then Overdrawn := true
```

Such a composite update is assigned a single timestamp; a site receiving the update executes it atomically. The effect of this is that the flag "Overdrawn" will be set to true if (and only if) Balance ever falls below zero in the timestamp-ordered execution of updates. A separate transaction can read this flag and perform the necessary compensation if Overdrawn is found to be true. This technique can be generalized to other applications, as described in [14].

Since updates issued by transactions may read the contents of the database (and their effects may be conditional on what they read), previous mutual consistency mechanisms that support only overwrites are no longer adequate. Our algorithm uses a *history* of all values of a data item written by all updates, not just the latest value and a timestamp. In addition, the algorithm records which data items were read by each update, so that higher-timestamped updates that conflict with a newly-arriving one can be identified; these conflicting updates are then undone and reexecuted to restore consistency. For the example update above, the arrival of any earlier-timestamped update that changes Balance will cause this update to be reexecuted. Overwriting updates are still executed efficiently; because such an update does not read anything, it is never reexecuted.

It is possible that the effect of reexecuting an update may be the same as it was on the previous execution of the update; in such a case, we say that the update was *unnecessarily* reexecuted. This cannot be avoided without using semantic information about what conditions on the update's readset (e.g., balance being sufficiently high) determine its writeset. An explicit objective of our algorithm design was to avoid the complexities of representing and manipulating semantic information about updates; this resulted in a conceptually simple algorithm that we were able to implement very quickly. The cost of an unnecessary update reexecution is not prohibitive, because it does not cause any "cascading" of further unnecessary reexecutions. That is, if reexecution of a given update has the same effect as before (which is determined by a run-time comparison), no higher-timestamped updates that read its output are marked for reexecution. Our algorithm appears to provide a sound framework for introducing optimizations based on semantic information, should it be necessary to further reduce unnecessary reexecutions; the treatment of specialized update operations in Section 4.1 is an example of such an extension.

The performance of our algorithm (which depends on the actual degree of conflict among updates) is a serious concern only if there is a prolonged network partition. While out of communication with some other site, a given site must retain history information as old as the timestamp of the last update received from that site. On reconnection, updates with very old timestamp may be received, possibly causing large numbers of higher-timestamped updates to be reexecuted. The storage and processing requirements of the algorithm therefore grow with the duration of the communication failure. For very long partitions (e.g., several days), these costs may be unacceptable. In addition, site clocks may drift too far out of synchrony for timestamp ordering to be meaningful. Therefore, mechanisms based on merging values rather than updates (such as [6,13]) may be more appropriate if a partition lasts an extremely long time; even these may not always be sufficient, and manual intervention may be needed.

The idea of undoing and reexecuting updates based on their conflicts to preserve the timestamp ordering property was previously introduced as part of the *log transformation* approach [3]. Log transformations were designed for a "discrete" network partition scenario, in which the analysis is performed just once to integrate updates issued on the different sides of a partition when all sites are fully connected again. The new SHARD architecture, on the other hand, does not assume that all sites are ever fully connected, and is thus able to handle a wider class of communication failures. New updates with old timestamps may arrive at a site at any time, and must be continuously integrated with already-processed ones. The mutual consistency algorithm must therefore be executed repeatedly, and it is important to retain as much information as possible for reuse. This is achieved using the history database, which is a novel contribution of our approach.

Our design of the history database is related to recent proposals for including the "time dimension" in a database [4,12]. To distinguish between the two, we will use the terms *history* to refer to a database (such as ours) that records past system states based on transaction time, and *historical* to refer to a database that represents perceived external states based on real-world time. When modeling real-world events in a historical database, it is not always the case that events are reported in the order that they occurred, because of varying external communication delays. In addition, there may be errors in reporting that need to be corrected later. Each event report, whose "timestamp" is the believed time of occurrence of the event, will typically contain one or a small number of basic facts that are directly installed in the database, and may in addition trigger the recomputation of some other data such as cumulative totals or flags indicating the presence of unusual transitions (such as an account being overdrawn). It is possible that the method we use for processing out-of-order updates will be applicable in historical databases as well: By recording which data items were read by the associated computations, it is possible to minimize the number of such computations that must be reexecuted when an old update arrives.

This paper describes the mutual consistency algorithm from the point of view of a single site that is receiving timestamped updates, issued by itself and by other sites, in some arbitrary order. The algorithm is executed by a module of the system called the *Checker*. A separate module, called the *Distributor*, is responsible for ensuring that all sites eventually receive all updates [1,7]. Even though sites may receive updates in different orders, the Checker algorithm at each site ensures that the end effect is as if all updates had been executed in timestamp order; eventual mutual consistency of database copies is therefore guaranteed. The current design and implementation of SHARD assumes a fully replicated database for simplicity. We are developing support for partial replication, by preanalyzing the update types permitted by the application and decomposing the database in such a way that every update can be executed locally by a site holding a copy of the affected data. The algorithm we present in this paper will apply to such a scenario as well.

The next section describes the model of data that we assume. Section 3 describes the basic Checker algorithm. Section 4 lists some extensions that we are exploring, including more efficient processing of specialized update operations (such as decrements). Section 5 concludes by summarizing the key features of the algorithm.

2. Model of Data

For the purpose of this paper, we assume that the database consists of a collection of named *objects* and their associated values, where an object is the smallest independently-updatable item of data. This is an abstract view of the contents of a database, not a user-oriented or implementation-oriented data model. Most if not all data models can be mapped into ours by appropriately restricting the set of possible object names and choosing related names for related objects. For example, in our prototype implementation of the algorithm we assume a relational database in which some set of fields in each relation comprise the *key* of the relation: no more than one tuple can have the same combination of values in the key fields, and the key fields of a tuple are not updatable. (To satisfy this restriction, it may be necessary to use internally-generated surrogates as keys.) Each non-key field of each tuple of each relation is viewed as a distinct "object", having as its name the triple *<relation-name,key-value,field-name>*.

The set of objects in the database does not change over time, i.e., is statically known from the initial database state. In the relational example, this requirement is met by pretending that every syntactically legal key value is present in the abstract counterpart of each stored relation, and making the following extensions (similar to other proposals for handling creation and deletion in historical databases [4]):

- For every relation, there is an additional boolean field named EXISTS that is implicit, i.e., not stored in the tuples of the relation. The value of the object <R1,K1,EXISTS> is true if the tuple with key K1 is present in relation R1, and is false if key K1 is not in R1.
- Every non-key field of every relation has a declared *default* value. If the tuple with key K1 is not present in relation R1, then the value associated with the object <R1,K1,F1> is the default value declared for field F1 in relation R1. The default value for a field may be a special "Undefined" or "null" value, or zero (which is appropriate for some numeric fields), or false (e.g., for the implicit field EXISTS), or any other application-specified value.
- "Deletion" of the tuple having key K1 in relation R1 is modeled abstractly as setting <R1,K1,F1> to the default value for every field F1 in R1; this includes the field EXISTS, whose value is set to false.
- "Creation" of a tuple with key K1 in R1 is modeled as setting <R1,K1,EXISTS> to true, and setting any other fields whose initial values are supplied; any field for which an initial value is not supplied retains its default value. To allow changes to tuples to be processed even if the creation of the

tuple has not been seen, we also treat tuple updates as setting EXISTS to true. If the application does not wish to treat tuple creation and tuple update identically, it can use more complicated update operations that test the value of EXISTS and then conditionally write field values.

3. The Checker Algorithm

This section describes how the Checker processes updates so that the effect on the database is as if the updates had been executed in timestamp order.

3.1 Database Updates

An *update* is a deterministic mapping from database states to database states. Each update issued is assigned a unique timestamp which is used both for totally ordering the updates and for identification. Descriptions (and timestamps) of all updates received by a site are recorded by the site so that an update can be reexecuted as needed to preserve consistency. We assume no particular representation and encoding for updates; we only assume that the description of an update can be executed by the Checker when needed.

The execution of an update results in a sequence of requests of the following kinds:

- *Read(object-name)*, which returns the value associated with the given object.
- *Write(object-name,value)*, which replaces the current value of the named object with the specified new value.

An update can perform arbitrary computations between requests, but its decision to issue a given Read or Write request can depend only on the values returned by previous Reads and on nothing else; this guarantees that updates are deterministic. We also assume for simplicity that an update does not Write the same object more than once, nor does it Read an object after Writing it. This is a minor restriction that can be satisfied by suitably rewriting complex updates. Specialized update operations such as increments and decrements must also be rewritten in terms of Reads and Writes. For example, the withdrawal of money from an account followed by the conditional setting of a flag (illustrated in Section 1) can be rewritten as:

```
newbal := Read(Balance) - amount
Write(Balance,newbal)   ;instead of Decrement
if newbal<0 then Write(Overdrawn,true)
```

While an update may read information from the database, it can only read individual objects and cannot issue general queries ("find all objects x such that P(x)"). (As stated earlier, queries issued by transactions are processed separately from updates.) We make this restriction mainly to avoid the problem of determining predicate overlap. The algorithm we present could be extended to handle updates that include queries, albeit inefficiently: when a query is issued by an update its "readset" is recorded as some worst-case superset that is guaranteed to always contain the set of objects satisfying the query.

3.2 Processing Old Updates

If an update arrives at a site out of timestamp order (i.e., later than some already-processed higher-timestamped updates), the following must be done in order to maintain a consistent database state that reflects all updates received as if they had been executed in timestamp order:

- For any objects that the update reads, the update must see the values of these objects "as of" its own timestamp; these values may no longer be the current ones.
- Object values that the update writes must also appear to have been written as of the update's timestamp; these may not have any effect on the current database state if some higher-timestamped update wrote the same object.
- If the update writes one or more objects that are read by a higher-timestamped update that has already been executed, then the effects (object values written) of the latter update may no longer reflect what it would have written if it had seen all earlier-timestamped updates executed in timestamp order. Therefore, such higher-timestamped updates need to be undone and reexecuted; on doing so, higher-timestamped updates that read their effects may also need to be undone and reexecuted, and so on.

To correctly process an update as of its timestamp, and to efficiently determine which updates need to be undone and reexecuted, we maintain a *history* of all object values written and all objects read by all updates that have been processed. Conceptually, this history is a collection of tuples of the following forms:

- *<Write,object-name,timestamp,value>*. These tuples are called *writeset entries*. No two writeset entries in the history can have the same object-name and timestamp. For every object there is one writeset entry in the history with the special timestamp *zero* (smaller than any other update's timestamp), that supplies the initial value of the object.
- *<Read,object-name,timestamp>*. These tuples are called *readset entries*. No two readset entries in the history can have the same object-name and timestamp.

We will use the following definitions in our discussion. Because update timestamps are unique, we use the timestamp T of an update to also refer to the update itself.

- *Writeset(T)* = {<x,v> | <Write,x,T,v> is in the history}
- *Readset(T)* = {x | <Read,x,T> is in the history}
- *Previous-Writer(x,T)* = the update that has the highest timestamp smaller than T among all updates that have x in their writesets.
- *Last-Writer(x)* = the update that has the highest timestamp among all updates that have x in their writesets. If we let *infinity* be a special value that is greater than any timestamp that can be assigned to an update, then Last-Writer(x) is the same as Previous-Writer(x,infinity).

- *Value-Upto(x,T)* = value v in the writeset entry <Write,x,Previous-Writer(x,T),v>. The "current" value of x in the database is Value-Upto(x,infinity).
- *Next-Writer(x,T)* = the update that has the lowest timestamp greater than T among all updates that have x in their writesets. If there is no such update (i.e., T>=Last-Writer(x)), then Next-Writer(x,T) is infinity.

To process a given update as of its timestamp T, Read and Write requests issued by the update are processed on the history as follows:

- Read(x) adds <Read,x,T> to the history and returns Value-Upto(x,T).
- Write(x,v) adds <Write,x,T,v> to the history.

Processing a Write request may require undoing and reexecuting higher-timestamped updates that read objects written by the given update T. It may also require that the current database copy be updated, if the latest value of the object in question is being changed; we defer discussion of this until Section 3.4. First we describe how the Checker algorithm determines which updates it must reexecute to maintain a consistent and correct history.

3.3 The Algorithm

The Checker algorithm is presented in Figure 1, in the form of a loop that runs forever installing the effects of received updates as quickly as possible. *New* updates are ones that arrived recently and have never been executed. An update that has been executed is initially marked *Valid* but may subsequently be marked *Invalid* if the execution (or reexecution) of an earlier-timestamped update causes the value of some object in its readset to change.

The algorithm in Figure 1 executes (or reexecutes) New and Invalid updates in timestamp order. While the Checker is running, additional New updates may be issued by the local site or received from other sites. An arriving New update may have lower timestamp than the current lowest-timestamped New or Invalid update (the update T being processed); this update will be processed in the next iteration of the loop.

The crux of the algorithm lies in how it selects higher-timestamped updates to be marked Invalid when an update T is first executed or is reexecuted. Suppose the update T is Invalid and is being reexecuted. The *changeset* of T is the set of objects that now have different values as a result of T's reexecution, and is the union of three disjoint sets:

1. The set of objects that T did not write in its previous execution and does write on reexecution, where the value written on reexecution is not the same as the value written by the previous (next lower-timestamped) writer of the object.
2. The set of objects that T did write in its previous execution and does not write on reexecution, where the value it wrote on the previous execution is not the same as the value written by the previous writer of the object.

```
while true do
  T := lowest-timestamped New or Invalid update,
     if none: wait for New updates to arrive, repeat
  ;initialize new readset and writeset
  newrset := empty
  newwset := empty
  execute update T, processing requests as follows:
    Read(x): add x to newrset
           return Value-Upto(x,T)
    Write(x,value): add <x,value> to newwset
  determine objects whose values changed:
    oldwset := Writeset(T)   ;empty if T New
    added := {x | <x,v2> in newwset
              and x not in oldwset
              and v2~=Value-Upto(x,T)}
    removed := {x | x not in newwset
              and <x,v1> in oldwset
              and v1~=Value-Upto(x,T)}
    changed := {x | <x,v2> in newwset
              and <x,v1> in oldwset
              and v2~=v1}
  changeset := union(added,removed,changed)
  for each x in changeset do
    ;mark future readers Invalid, through next writer
    for each update U such that U is Valid
           and U>T and U<=Next-Writer(x,T)
           and x in Readset(U):
      mark U Invalid
    end for
    if T>=Last-Writer(x) then
      update x in current database copy
  end for
  ;update history, atomically
  replace Readset(T) in history with newrset
  replace Writeset(T) in history with newwset
  ;done with T
  mark T Valid
end while
```

Figure 1. The Checker Algorithm

3. The set of objects that T wrote both in its previous execution and in its reeexection, where the values written in the two executions are not the same.

If the update T is New and is being executed for the first time, then it did not write anything previously (in Figure 1, "oldwset" is empty). Its changeset has only one component, corresponding to the first case above.

If a given object x is in T's changeset, then higher-timestamped readers of x must be marked Invalid and scheduled for reexecution. However, if some later update, Next-Writer(x,T), also wrote x, then readers of x having timestamp greater than Next-Writer(x,T) need not be marked Invalid because the value they would read is the one written by Next-Writer(x,T) (or by another higher-timestamped writer of x) and is unaffected by T's change to x.

That the algorithm above in fact gives the correct result can be verified as follows. For each object x in Readset(U), we define *Value-Seen(x,U)* to be the value of x that was read by U when it was last executed. This is the value that Value-Upto(x,U) had in the history at the time U was executed. So long as

Value-Upto(x,U) equals Value-Seen(x,U) for every x in Readset(U), the effects of U recorded in the history are still valid and U need not be reexecuted. We observe that, on executing (or reexecuting) a given update T, the Checker algorithm marks a Valid update U Invalid if and only if the execution of T caused Value-Upto(x,U) to change for some x in Readset(U). Therefore, if an update U is Valid, it must be the case that Value-Upto(x,U) still equals Value-Seen(x,U) for every x in Readset(U). Any update U for which this is not true must have been marked Invalid, and will be reexecuted by the Checker.

While updates whose readsets have been modified will be reexecuted (guaranteeing correctness of the algorithm), efficiency requirements dictate that no update be reexecuted unless its readset has been modified. This will be true except in cases where Value-Upto(x,U) has changed more than once since the last time U was executed and happens to be equal to Value-Seen(x,U) again. In this unlikely event, U will be unnecessarily reexecuted and will have the same writeset as before. Such unnecessary reexecutions could be avoided if Value-Seen were recorded in the readset entries of updates. We chose to save the cost of storing these values (instead recording only the names of objects read), and of comparing them against Value-Upto before reexecuting an update, with the expectation that the cost of a few unnecessary reexecutions would not be significant in comparison.

3.4 Updating the Current Database

Assume that a separate copy of the current database state is being maintained to service queries from users and application programs. A change to the history should cause the database copy to be updated if the highest-timestamped writeset entry for an object, say x, is being added or removed or replaced and the object's current value is actually changing as a result, i.e., the object x is in the update T's "changeset" and T>=Last-Writer(x). If the highest-timestamped Write of x is being removed on reexecution of T, the current database copy should be updated to reflect the value written by the previous writer of x; if T is writing a new current value of x, that value should be installed in the current database copy.

3.5 Prototype Implementation

The Checker algorithm has been implemented for a relational database. As described in Section 2, individual fields (including the implicit EXISTS field) of tuples are treated as independently-updatable objects, with names of the form <relation-name,key-value,field-name>. The current database state is stored in INGRES relations.

The history information needed by the algorithm is stored in UNIX[tm] files, rather than in the DBMS, in order to support the Checker's special processing requirements. The data structures we use are:

- A file containing the update descriptions and timestamps; updates are asynchronously appended to this file as they arrive.
- A writeset file that records names and values of objects written by updates.
- A readset file that records names of objects read by updates.
- An update status file, with entries linked in timestamp order, that records each update's status (New or Valid or Invalid) and contains pointers to the update's description and to the readset and writeset entries of the update.

Even though individual fields of a tuple can be updated independently, we assume that many updates will read or write more than one field of a tuple. Such accesses to multiple fields of the same tuple are grouped into a single Read or Write request. The resulting readset or writeset entries are stored as a single record in the corresponding file. Writeset records from different updates for the same tuple are chained in timestamp order, to permit efficient searching for the values of any group of fields of the tuple as of any desired timestamp. Similarly, readset records for the same tuple are chained in timestamp order, so that higher-timestamped readers of any group of fields of the tuple can be found in a single scan. We have not attempted further optimizations of the data structures, because the Checker is not currently a bottleneck in the system. It would probably be useful to cluster readset and writeset records by timestamp, to improve locality and to make garbage-collection of old information (Section 4.3) more efficient.

We have not yet made the Checker implementation fully crash-resistant. What we have done (reflected in Figure 1) is order the different steps of an update's execution or reexecution (installing the new readset and writeset, and marking updates Invalid or Valid) in such a way as to make the execution restartable. That is, a crash between steps will at worst cause an unnecessary reexecution of an update; it will not destroy the integrity of the history information. Recovering from a crash that occurs during execution of a step will require performing each step as an atomic recoverable change, using known techniques such as intentions lists or shadow pages. Breaking the processing of an update into multiple steps will also allow concurrent processing of multiple updates, as discussed in Section 4.4.

4. Extensions

This section outlines several extensions that we are considering for the Checker algorithm. While these have not been worked out in detail, our intention in describing them is to illustrate that the basic algorithm and the history database provide a sound and robust foundation for introducing a variety of enhancements to the performance and functionality of the system.

4.1 Specialized Update Operations

Certain types of objects permit update operations (such as increment and decrement) other than a Write operation that replaces the current value of an object

with a given new value. When such operations are translated into Read and Write requests, the arrival of a decrement operation after a higher-timestamped one will cause the latter to be reexecuted (since the latter's readset is being changed). While this gives the correct final result, the reexecution would not have been necessary if the Checker could take advantage of the commutativity of decrement operations.

To provide better support for specialized update operations, our history database can be extended to allow such operations to be stored directly in the history without executing them. The history information for an object may then contain a mixture of Writes and other operations. The computation of the value of an object is deferred until it is actually needed. That is, when a Read request is issued for a given object x as of timestamp T, the desired Value-Upto(x,T) is computed by finding the most recent Write of x (with highest timestamp smaller than T) and then executing in timestamp order all intervening operations on x.

If the current value of an object is being maintained separately (in the current database copy), it must be kept up-to-date as Writes or other operations are added to or removed from the object's history. In the worst case, the new current value will have to be recomputed as described above. However, there are many special cases where the new value can be computed more efficiently:

- An operation is added, with higher timestamp than the last Write, that commutes with all higher-timestamped operations (e.g., all are increments or decrements): simply execute the operation on the current value. This case applies trivially if there are no other higher-timestamped operations.
- An operation is removed (because the Invalid update that had issued the operation did not do so on reexecution), with higher timestamp than the last Write, that is *invertible* and commutes with all higher-timestamped operations (if there are any): execute the inverse operation (e.g., decrement instead of increment) on the current value.

4.2 Query Consistency

If a transaction issues a query against the current database copy, there is no guarantee that it will see a consistent database state reflecting the timestamp-ordered execution of some set of updates. The only time the database is consistent is when there are no Invalid updates waiting to be undone and reexecuted. The Checker could defer installing changes to the current database copy until there are no Invalid updates (i.e., the last remaining one is reexecuted); transactions would then always see consistent information. However, the information seen would not reflect updates received since the last time there were no Invalid updates. It is possible that some transactions might prefer to see the latest information even if it is inconsistent; these would be unnecessarily penalized. Therefore, it seems preferable to install changes directly in the current database, and let each transaction decide whether or not it needs to see consistent information. The Checker can make information about the status of updates available for transactions to examine; a transaction that does wish to see consistent information simply waits until there are no Invalid updates.

If a transaction wishes to wait until all Invalid updates are reexecuted, this may take a very long time if new updates with old timestamps continue to arrive and keep causing existing updates to be marked Invalid. Availability could be improved if a transaction only had to wait for that part of the database in which it is interested to become consistent. It should be possible to extend the Checker algorithm so that when an update is marked Invalid a recursive scan is performed of updates that read its output and updates that read their output and so on, marking the affected data items in the current database copy as potentially inconsistent. Some additional effort will be needed to later "unmark" such a potentially inconsistent data item, if reexecution of the update that wrote it has the same effect as before.

Further improvements would be possible if the Checker could selectively schedule New and Invalid updates for execution (or reexecution) based on query requirements, rather than treating all updates as equal and always executing them in timestamp order. The correctness of the algorithm would not be affected if updates were executed in some other order. Certain classes of queries could be favored by statically assigning priorities to different types of updates and executing higher-priority updates before lower-priority ones. It may also be useful to decompose updates into independent pieces that can be assigned different priorities, e.g., computation of the flag Overdrawn could be deferred when an account's Balance is updated. A more dynamic mechanism would select New and Invalid updates for execution based on the data requirements of the queries outstanding (and anticipated) at any given time; we are currently looking at what additional information the Checker would need in order to do this.

4.3 Garbage-Collection

The history database can grow without bound as more and more updates arrive; this will eventually exhaust the available storage at a site. It is possible to discard old information from the history once it has been determined, by agreement among all sites, that no further updates with timestamps smaller than a specified "cutoff" will be issued in the future. Once a site has determined a cutoff timestamp, say T0, the update descriptions and readset entries of updates with timestamp smaller than T0 can be discarded from the site's history database. The writeset entries of these updates can also be discarded, except that a newly-arriving update (which can have timestamp as small as T0) may need to read the value of an object as far back as T0. Therefore, the site must retain Value-Upto(x,T0) for each object x (this is in effect a "checkpoint" of the database as of timestamp T0); all earlier object values can be discarded.

4.4 Internal Concurrency

The Checker algorithm we have described executes updates one at a time. This was done mainly for ease

of exposition and to avoid concurrency issues in the implementation. The algorithm would continue to work correctly if updates were executed concurrently, with appropriate low-level synchronization of access to the history data structures. The execution (or reexecution) of a New or Invalid update could be started immediately whenever some other update's execution is waiting for disk I/O and a processor is available. This would make information available for queries sooner, but would also increase the likelihood that an update will be undone and reexecuted because a concurrently-executing earlier-timestamped update may invalidate its readset. In the extreme, if a processor could be allocated to every update, the behavior of the algorithm would be very similar to Jefferson and Motro's "Time Warp" mechanism [9]. The Time Warp approach and our strict sequential approach can be viewed as opposite extremes on the spectrum of optimism versus pessimism. In future research, we hope to explore other points along this spectrum, to achieve a balance between concurrency and cost of reexecution.

5. Summary

We presented an algorithm that processes updates arriving out of timestamp order in such a way that the database state always reflects updates received as if they had been executed in timestamp order. The algorithm uses a history of values written and objects read by all updates, and schedules updates for reexecution based on their conflicts with newly-arriving earlier-timestamped updates. The algorithm dynamically determines the difference between the effects of an update's previous execution and its reexecution in order to prevent the cascading of unnecessary reexecutions of updates.

The Checker algorithm has been implemented for a relational database in the SHARD prototype at CCA, and its performance under initial testing appears reasonable. We outlined several promising extensions to the basic algorithm, suggesting that the technique provides a reasonable framework for adding future enhancements. It is possible that our technique may be useful outside its original scenario, in a historical database where "timestamps" refer to real-world time rather than system clock time and updates may arrive with old timestamp because of external delays or because a prior error needs to be corrected. We leave this question open for future research.

Acknowledgments: We would like to thank Barbara Blaustein for many discussions about the log transformation technique which provided the background and inspiration for this algorithm, and Arnon Rosenthal and the referees for their comments and suggestions.

References:

[1] B. Awerbuch and S. Even, "Efficient and Reliable Broadcast is Achievable in an Eventually Connected Network," *Proc. Symp. Principles of Distributed Computing*, 1984, 278-281.

[2] A.D. Birrell, R. Levin, R.M. Needham, and M.D. Schroeder, "Grapevine: An Exercise in Distributed Computing," *Comm. ACM 25*, 4 (April 1982), 260-274.

[3] B.T. Blaustein and C.W. Kaufman, "Updating Replicated Data during Communications Failures," *Proc. Eleventh Int. Conf. Very Large Data Bases*, August 1985, 49-58.

[4] J. Clifford and D.S. Warren, "Formal Semantics for Time in Databases," *ACM Trans. Database Systems 8*, 2 (June 1983), 214-254.

[5] M.J. Fischer and A. Michael, "Sacrificing Serializability to Attain High Availability of Data in an Unreliable Network," *Proc. Symp. Principles of Database Systems*, 1982, 70-75.

[6] H. Garcia-Molina, T. Allen, B. Blaustein, R.M. Chilenskas, and D.R. Ries, "Data Patch: Integrating Inconsistent Copies of a Database after a Partition," *Proc. Third Symp. Reliability in Distributed Software and Database Systems*, October 1983.

[7] H. Garcia-Molina, N. Lynch, B. Blaustein, C. Kaufman, S. Sarin, and O. Shmueli, "Notes on a Reliable Broadcast Protocol," CCA technical report, 1985.

[8] D.K. Gifford, "Weighted Voting for Replicated Data," *Proc. Seventh Symp. Operating Systems Principles*, November 1979, 150-162.

[9] D. Jefferson and A. Motro, "The Time Warp Mechanism for Database Concurrency Control," *Proc. Int. Conf. Data Engineering*, February 1986, 474-481.

[10] P.R. Johnson and R.H. Thomas, "The Maintenance of Duplicate Databases," ARPA Network Working Group Request for Comments (RFC) 677, Bolt Beranek and Newman Inc., January 1975.

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

[12] V. Lum, P. Dadam, R. Erbe, J. Guenauer, P. Pistor, G. Walch, H. Werner, and J. Woodfill, "Designing DBMS Support for the Temporal Dimension," *Proc. ACM SIGMOD Annual Conf.*, 1984, 115-130.

[13] D.S. Parker, G.J. Popek, G. Rudisin, A. Stoughton, B.J. Walker, E. Walton, J.M. Chow, D. Edwards, S. Kiser, and C. Kline, "Detection of Mutual Inconsistency in Distributed Systems," *IEEE Trans. Software Engineering SE-9*, 3 (May 1983), 240-246.

[14] S.K. Sarin, "Robust Application Design in Highly Available Distributed Databases," *Proc. Fifth Symp. Reliability in Distributed Software and Database Systems*, January 1986, 87-94.

[15] S.K. Sarin, B.T. Blaustein, and C.W. Kaufman, "System Architecture for Partition-Tolerant Distributed Databases," *IEEE Trans. Computers C-34*, 12 (December 1985), 1158-1163.

[16] R.H. Thomas, "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases," *ACM Trans. Database Systems 4*, 2 (June 1979), 180-209.