Asynchronous Version Advancement in a Distributed Three Version Database

H. V. Jagadish
AT&T Laboratories
jag@research.att.com

Inderpal Singh Mumick
AT&T Laboratories
mumick@research.att.com

Michael Rabinovich
AT&T Laboratories
misha@research.att.com

Abstract

We present an efficient protocol for multi-version concurrency control in distributed databases. The protocol creates no more than three versions of any data item, while guaranteeing that (1) update transactions never interfere with read-only transactions, (2) the version advancement mechanism is completely asynchronous with (both update and read-only) user transactions, and (3) read-only transactions do not acquire locks and do not write control information into the data items being read. This is an improvement over existing multi-versioning schemes for distributed databases, which either require a potentially unlimited number of versions, or require coordination between version advancement and user transactions. Our protocol can be applied in a centralized system also, where the improvement over existing techniques is in reducing the number of versions from four to three.

The proposed protocol is valuable in large applications that currently shut off access to the system while managing version advancement manually, but now have a need for automating this process and providing continuous access to the data.

1 Introduction

Databases often maintain multiple versions of data to avoid interference between read-only queries and update transactions. Many multi-versioning concurrency control protocols have been proposed, both for centralized and distributed databases (see, *e.g.*, [SR81, BHR80, CG85, Wei87, CFL$^+$82, MPL92, AS89, BC92]). Of special interest to us are protocols that satisfy the following three properties (*called the non-interference requirement*): (1) completely decouple read-only queries from updates, (2) never delay transactions on account of version management, and (3) do not require read-only queries to acquire any locks or write any control information to data items they access (such writes basically turns the read-only queries into much heavier-weight updates). At the same time, while read-only queries should not fall too far behind the updates, it is not critical that the queries read the latest committed data.

This non-interference requirement eliminates from consideration many protocols, like the two-version protocols of [BHR80, SR81] where interference between read and update transactions is still possible, and the protocol of [Wei87] that requires read transactions to write *read timestamps* to data items they access.

Existing techniques that satisfy the non-interference requirement fall into two categories. Techniques belonging to the first category (see [CFL$^+$82, BC92] for centralized schemes and [CG85] for a distributed scheme) may create a potentially unlimited number of data versions due to a long-running read transaction. Read transactions incur the overhead of following a potentially unlimited chain of pointers to data versions to determine the version they can use. Moreover, in the distributed case [CG85], they have to know in advance the set of sites they will visit.

Techniques in the second category [MPL92, WYC91] introduce a separate *version advancement* procedure to move from one version to another, and require four versions to satisfy the non-interference requirement in a centralized database. However, the extension to a distributed database discussed in [MPL92] requires version advancement to be coordinated with user operations: otherwise, autonomous version advancement may cause user operations to be aborted. Thus, the non-interference requirement is violated in the distributed case.

We propose a new scheme, called the asynchronous version advancement protocol with at most three versions, or AVA3 protocol for short, that satisfies the non-interference requirement with only *three* versions in a distributed database. Further, one of the three versions is transient, and the protocol is rather simple in comparison to existing schemes. Our scheme requires that two-phase locking is used for concurrency control of update transactions. Read transactions do not acquire any locks: they are serialized with respect to updates by using appropriate data versions.

1.1 Our Approach

Our approach grew out of the observation of manual version management routinely performed by database administrators in real life. The key version advancement idea is intuitively developed in this sub-section.

In manual versioning, the data is kept in two separate copies, one for operational support and one for read-only queries (decision support, auditing, etc.). Periodically, updates accumulated in the operations support copy are flushed to the read-only copy of the data. Access to the read-only data is blocked during the flushing process.

Typical examples of manual versioning are database applications for telephone calls or credit card transactions, where it is used to decouple customer access to telephone call (or credit card) transactional data from the support for completing, processing, and collecting telephone call (or credit card) transactions. The read-only copy is used for customer inquiries while the operations support copy is used to record current calls and transactions activity.

The AVA3 protocol achieves the same functionality as above, but without requiring two full copies of the data, and without blocking access to the read-only copy. The protocol normally maintains one full copy of the database and one partial (perhaps empty) update copy, to which all update transactions write any data objects they modify. Read transactions access the main copy. Update transactions read either the update copy of the data (if it exists) or the main copy (otherwise). Periodically, a *version advancement* protocol runs to make newer data available to reads. Version advancement may be triggered by the system due to a variety of reasons. For instance, we may want to advance versions every hour, or once a certain number of update transactions have accumulated, or when the difference in value of data items in the read and update copies exceeds some threshold, or after an update transaction of a particular kind commits.

Each node keeps two numbers – a query version number and an update version number, corresponding to the read and (possibly non-existent) update copies of each data item at the node. The version advancement protocol starts by asynchronously promoting the update version number on each node. Update transactions that start on a node after it promoted its update version number write modified data into a new third copy. During a given instance of version advancement execution, an update transaction T that started on a node before its update version number was advanced continues to use the second copy, *until it needs to access a data item for which the third copy already exists.* (This can happen if the data item has been updated by a concurrent transaction

S that started and committed after the node had incremented its update version number.) In this case, transaction T switches to the third copy. To do that, all updates already executed by this transaction must be transferred from the old (second) copy to the new (third) copy. We introduce a moveToFuture function that takes care of this without restarting the transaction or obtaining any locks.

Similarly, at commit time, it may be discovered that an update transaction used different version numbers at different nodes. We deal with such cases by modifying the steps of the two phase commit protocol: with the prepare message, each subtransaction includes the version number it used by the time it reached the prepared state. The commit coordinator takes the maximum of the reported version numbers and distributes it with the commit message. When a participant receives the commit message with a version number exceeding the one used by the subtransaction, it performs the moveToFuture function.

Once all nodes promote their update version number, no version mismatch can occur for new update transactions. As soon as existing update transactions, which started with the old update version number, all finish, the second copy of the database on all nodes is guaranteed to be mutually consistent, and read transactions can be allowed to use this copy. This is done by promoting, again completely asynchronously, the read version number on all nodes. A read transaction that starts on a node after it promoted its read version number uses the second copy for all its data accesses. If at the start time, the node still has the old read version number, the first copy is used. When it is detected that all read transactions using the first copy of the data have finished, this copy can be garbage-collected with no danger that it will be needed. This completes the version advancement process, leaving the system with at most two copies of the data again.

1.2 Results

We propose the AVA3 protocol for distributed databases that uses at most three versions to decouple read and update transactions while guaranteeing that (1) read transactions never interfere with update transactions, (2) the version advancement mechanism is completely asynchronous with user transactions, and (3) read transactions do not require any locks and do not update any information in the data items they access. The read transactions are allowed to increment some main memory counters associated with the node using latches (no locks).

For the centralized case (that is, when there is only one site) the AVA3 protocol reduces the number of versions needed to guarantee the non-interference requirement from four versions [MPL92, WYC91] to three versions.

Finally, our protocol is extremely simple and light-weight. It does not require maintaining and manipulating the completed transaction lists as in [CG85], or lists of update transactions that accumulated since last version advancement as in [MPL92], or a transaction dependency graph of [BHR80]. Instead, our protocol maintains only a few integers per node and two integers per transaction, in addition to standard data structures for accessing data in desired versions.

Our protocol achieves these results at the expense of allowing read transactions to access a stale snapshot of the data. By running frequent version advancements in the background, the staleness of the snapshot used by a read transaction Q can be limited to the age of the longest-running read transaction that was executing at the time Q started.

2 System Model and Notation

We assume that update transactions use strict two-phase locking, *i.e.*, they lock every data item they access on a node. They release shared read locks upon sending the prepare message to the coordinator of the commit, and release exclusive write locks upon receiving a commit message from the coordinator. A shared lock on a data item means that other update transactions can read but not write *any version of* the data item. An exclusive lock on a data item means that no other update transaction can read or write to any version of the data item. Query transactions do not obtain any locks, so they can read appropriate versions of a data item that is locked by an update transaction.

The model of user transactions follows the R* system [MLO86]. An update transaction is submitted to one server first, which becomes the root of the transaction. It executes a root subtransaction and then sends children subtransactions to other nodes if necessary. When a subtransaction is done, the server sends a *Prepared* message to the parent, and waits for the *Commit*. *Prepared* messages propagate to the root, which makes a decision on whether to commit the transaction. If the decision is to commit, it sends the *Commit* message to the nodes to which it sent subtransactions. These nodes commit their subtransactions, and send the *Commit* message further down.

Read transactions (queries for short) also follow the tree model of execution. A query is submitted to the root, which sends descendent subqueries to other nodes. When a subquery completes, it waits for the results of its descendant subqueries, incorporates them into its own results and sends them to its parent and commits. When a root subquery and all its children subqueries are finished, the root subquery returns the results to the user.

We assume there are n nodes in the distributed system, labeled $1, \ldots i, j, k, \ldots n$. Upper case letters Q, S, and T refer to transactions in the system. The subtransaction of transaction T at node i is referred to as T_i. We use lower case letters, x, y, and z to refer to data items. $x(v)$ denotes the value of x in version v, if x exists in version v, else it is undefined. $\mathsf{maxV}(x)$ denotes the maximum version in which data item x exists.

3 The AVA3 Protocol

The asynchronous version advancement (AVA3) protocol builds upon two phase locking for local concurrency control and two phase commit for distributed commitment. The protocol specifies the versions to be accessed by each read and update subtransaction, how to advance the versions so that more current data is read by subsequent subtransactions, and how to garbage collect old versions of data items.

Each data item can exist in at most three versions. We assume here, for simplicity, that version numbers increase monotonically with time. An implementation could re-use old version numbers, employing only three distinct numbers. We assume that data structures and indices are built so that the following questions can be answered efficiently: (1) Does data item x exist in version v? and (2) What is the maximum existing version of x? Techniques to build such indices are discussed in [MPL92].

3.1 Control State and Subroutines

Control state: Each site i maintains the following three variables:

- u_i, an *update version number*. This is the version of data to be updated by any new update subtransaction at node i.

- q_i, a *query version number.* This is the version of data to be read by any new query starting at node i.
- g_i, a *garbage version number.* This is the version number of data that can or has been garbage-collected.

Site i also maintains two counters for each active version number it has, a query counter $\mathsf{queryCount}(i, v)$, and an update counter $\mathsf{updateCount}(i, v)$. These counters (together referred to as *transaction counters*) indicate the number of query or update subtransactions in progress at node i that started execution by accessing data in version v. Transaction counters are maintained in main memory only. This is important, since it ensures that updates to these counters are lightweight, involving no disk access, logging, or even locking. In consequence, upon a system crash, these counters are lost. But this creates no problem because they can start out with an initial value of 0 upon after-crash recovery, since all transactions in progress are aborted in the recovery process.

Each query and update subtransaction T_i maintains a version number $V(T_i)$, called the version of T_i, which specifies the version of the data currently being accessed. For update subtransactions, $V(T_i)$ may change during the execution. Consequently, each update subtransaction also maintains the starting value of $V(T_i)$, denoted $\mathsf{startV}(T_i)$, and called the starting version of T_i. Version numbers associated with a subtransaction are also maintained in main memory only.

Thus, on a node i, the query counter $\mathsf{queryCount}(i, v)$ is equal to the number of active query subtransactions for which $V(Q_i) = v$. The update counter $\mathsf{updateCount}(i, v)$ is equal to the number of active update subtransactions for which $\mathsf{startV}(T_i) = v$.

At start-up time, all records exist in a single version 0, and for each node i, $q_i = 0$, $u_i = 1$, $\mathsf{updateCount}(i, 0) = \mathsf{updateCount}(i, 1) = \mathsf{queryCount}(i, 0) = \mathsf{queryCount}(i, 1) = 0$.

Deletions of data items: Deletion of a data item y in version v is modeled by simply marking y as deleted in version v, without removing the object, unless v is the only version of the object in the system. Subsequently, when all earlier versions of y are garbage-collected from the system, the data item y can be removed from version v as well.

moveToFuture: To switch an active update subtransaction T_i from accessing version $\mathsf{oldV} = V(T_i)$ to a new version newV, at some node i, we define a function $\mathsf{moveToFuture}(T_i, \mathsf{newV})$. The purpose of this function is to bring the database at node i to a state the database would have if T_i never touched any data items in version $\mathsf{oldV}\}$, and instead had operated all along with version newV. The implementation of this function is described in Section 4. The important point to note now is that the implementation does not involve aborting and replaying T_i, or acquiring any locks. Therefore, it never delays nor is delayed by other transactions. In other words, moveToFuture does not introduce hidden synchronization between version advancement and user transactions.

3.2 Version Advancement and Garbage Collection

Assume that reads are occurring on a common query version oldq, and updates on a common update version oldu at each node in the distributed database. Version advancement is meant to move the system forward by creating a temporary third version.

Any node can initiate version advancement, thus becoming the coordinator of version advancement, provided it does not participate in version advancement already. Correctness of the protocol does not depend on having a single coordinator - it is possible for multiple nodes to initiate version advancement independently. However, we guarantee that all coordinators are advancing the system to the same new update and read version numbers. In other words, the system cannot be in the process of advancing to different versions at the same time.

The basic strategy for version advancement is to make new update transactions update a new version $\mathsf{newu} = \mathsf{oldu} + 1$, wait for old update transactions on version oldu to finish, then make new query transactions read data in version $\mathsf{newq} = \mathsf{oldu} = \mathsf{oldq} + 1$, wait until queries on version oldq complete, and finally garbage collect version oldq.

A node k may initiate version advancement only if $u_k = g_k + 2$ and all data in version g_k has been garbage-collected, which indicates it is not in the process of version advancement already. Let $\mathsf{oldu} = u_k$, $\mathsf{oldq} = q_k$, and $\mathsf{oldg} = g_k$ be update, read, and garbage version numbers before k starts version advancement. Version advancement is done in three phases:

Phase 1. Switching to a new update version: Node k determines the new update version number $\mathsf{newu} = u_k + 1$, and sends a message *advance-u* with the new update version number newu to every node, including itself.

Each node i receiving message *advance-u*(newu) executes the following steps:

```
IF (u_i > newu)
    ignore and terminate
IF (g_i < newu − 3)
    g_i = newu − 3
    garbage-collect data of version g_i (see Phase 3)
IF (u_i < newu)
    u_i = newu
    updateCount(i, newu) = 0
wait until updateCount(i, newu − 1) = 0
send ack-advance-u(newu) to node k
```

Each node i receiving this message compares its update version number, u_i, with newu. If $u_i > \mathsf{newu}$, this version advancement message is obsolete (i already advanced its version beyond that), and the message is simply ignored.

If $u_i < \mathsf{newu}$, node i sets $u_i = \mathsf{newu}$, so that all subsequent update subtransactions at node i execute on version newu), and allocates and initializes the update counter $\mathsf{updateCount}(i, \mathsf{newu}) = 0$. If $u_i = \mathsf{newu}$, it means that node i has already initiated version advancement to newu (because it heard about advancement already from some other node [Section 3.4], or because a different coordinator also initiated this version advancement).

In either case, node i waits until all local update subtransactions that started before i switched to newu finish. Node i detects that all these subtransactions finished when $\mathsf{updateCount}(i, \mathsf{newu}-1) = 0$. Once this condition is true, i sends an acknowledgment *ack-advance-u* to the coordinator. To identify which version advancement operation is being acknowledged, the acknowledgement carries newu.

Also, if i's garbage version number is less than $\mathsf{newu} - 3$, this means that i has not finished Phase 3 of another instance of version advancement with coordinator different from k. In this case, i infers that k must have already gargabe-collected all versions up to $\mathsf{newu} - 3$ (otherwise, it could not have initiated new version advancement), and therefore it is safe for i to do so also. Thus, i sets $g_i = \mathsf{newu} - 3$ and garbage-collects this version of the data (see Phase 3).

The coordinator collects acknowledgments from all the nodes. Once every node has confirmed initiating version advancement, it is guaranteed that (a) all new update subtransactions will update data in version newu and (b) all updates on version oldu are complete, and version oldu is stable. The coordinator now initiates Phase 2.

Due to the possibility of multiple coordinators, k may act as the coordinator of one version advancement operation and a participant in other instances of the protocol execution. Therefore, while waiting for *ack-advance-u*(newu), k may receive a message *advance-q*(newq) from another coordinator executing Phase 2 of this protocol. In this case, k stops executing the current protocol and ignores any *ack-advance-u*(newu) messages, both the ones already received and the ones it may receive in the future.

Phase 2. Switching to a new query version. The coordinator determines the new query version number $\mathsf{newq} = q_k + 1$ and sends a message *advance-q* with the new query version number newq to every node, including itself.

Each node i receiving this message compares its q_i with newq. Similar to Phase 1, i ignores the message if $q_i > \mathsf{newq}$.

If $q_i = \mathsf{newq}$, node i has already switched to a new query version. If $q_i < \mathsf{newq}$, node i increments q_i to be equal to newq, so that all new queries at node i are executed against the version newq, and allocates and initializes the query counter $\mathsf{queryCount}(i, \mathsf{newq}) = 0$. In either case, i waits for for all existing queries that are reading data of version $\mathsf{oldq} = \mathsf{newq} - 1$ to terminate by checking if $\mathsf{queryCount}(i, \mathsf{oldq}) = 0$, and sends an acknowledgment *ack-advance-q*(newq) to the coordinator.

Once all acknowledgements arrive, the coordinator starts Phase 3. Again, while waiting for *ack-advance-q*(newq) messages, k may receive a message *garbage-collect*(newg) from another coordinator executing Phase 3 of this protocol. In this case, k stops executing the current protocol and ignores any *ack-advance-q*(newq) messages that it has received or will receive in the future.

Phase 3. Garbage Collection. The coordinator sends a *garbage-collect*(newg) message to all nodes. Upon receiving the garbage collection message, a node i compares newg received with its garbage version number, g_i. If $g_i \geq \mathsf{newg}$, this message is ignored. Otherwise, node i sets $g_i = \mathsf{newg}$ and garbage-collects the data of this version as follows. For every data item x, if x exists in version $q_i = \mathsf{newq}$, node i garbage-collects the earlier version $g_i = \mathsf{newg} = \mathsf{oldq}$ of x. If data item x does not exist in version newq, the node i changes the number of oldq version of x to version newq. Further, node i deletes the query counter for version oldq and the update counter for version newq.

Note that once i completes garbage collection of version newg, it can again initiate version advancement. If it does, it will have $\mathsf{newu}' = u_i + 1 = \mathsf{newu} + 1$. At this point, some nodes that participate in "old" version advancement initiated by k may have not yet received *garbage-collect*(newg) message from k. If a node j receives message *advance-u*(newu') from i before message *garbage-collect*(newg) from k, node j may still have query version $\mathsf{oldg} < \mathsf{newu}' - 3$. As already mentioned in Phase 1, j infers in this case that i must have already garbage-collected all versions up to $\mathsf{newu}' - 3$ and therefore it is safe for j to do so also.

3.3 Read-Only Transactions

The protocol for executing a subquery Q_i of query Q on node i is as follows.

1. If Q_i is the root, execute the following two steps atomically:
   - Set $V(Q_i) = q_i$.
   - Increment query counter $\mathsf{queryCount}(i, V(Q_i))$ indicating that one more query subtransaction is executing against version $V(Q_i)$.
2. If Q_i is not a root, it is assigned a version number $V(Q_i)$ by its parent (see Step 4 below). Then, simply increment the query counter $\mathsf{queryCount}(i, V(Q_i))$.

If $V(Q_i) > q_i$, it means that version advancement to the new query version has begun, but node i has not yet received the *advance-query-version* message from the coordinator. In such a case, node i initiates advancement of the query version (sets $q_i = V(Q_i)$, and initializes $\mathsf{queryCount}(i, V(Q_i)) = 0$) before doing the above actions.

3. If Q_i reads a data item x, read the maximum existing version of x not exceeding $V(Q_i)$. No locks are obtained (a short-term latch may be obtained).

4. If Q_i creates child subtransactions, assign them a version number equal to $V(Q_i)$, and send them to their respective destination nodes.

5. When all data items have been read and the results from all descendant subqueries have arrived, compose the results of Q_i, and sends them to the parent (if Q_i is a non-root), or to the user (if Q_i is the root), decrement $\mathsf{queryCount}(i, V(Q_i))$, and commit.

3.4 Update Transactions

The protocol is the same for root and descendant update subtransactions. A subtransaction T_i starts at node i by looking up the update version number u_i at the node, and using it as its starting and current version number. While subtransaction T_i is executing, it is possible that a version advancement process increments the update version number at node i. An update subtransaction S_i starting later may thus have a starting version number $\mathsf{startV}(S_i) = \mathsf{startV}(T_i) + 1$, and may update a data item x in version $\mathsf{startV}(S_i)$ and commit. If T_i later wants to access data item x, it must be serialized after subtransaction S_i. We accomplish this by moving all updates of T_i from its current version into a new version v (using the moveToFuture function) any time T_i accesses a data item that exists in version $v > V(T_i)$. The current version of the transaction is also advanced to v.

Another complication is that while subtransaction T_i updates on version u_i at node i, another subtransaction T_j of the same transaction T may perform updates on version $u_j = u_i + 1$ at node j, if node j has started version advancement, but node i has not started version advancement. We catch this discrepancy at commit time, during the two phase commit protocol. We then make the whole transaction T commit in version u_j, which means that all updates performed by T_i in version $u_i = u_j - 1$ are moved up to version u_j (using the moveToFuture function).

With this background, we list the protocol below:

1. When starting a subtransaction T_i, a site i executes the following two steps atomically:
   - Set $V(T_i) = \mathsf{startV}(T_i) = u_i$.
   - Increment the update counter $\mathsf{updateCount}(i, \mathsf{startV}(T_i))$ for version $\mathsf{startV}(T_i)$.

2. To read a data item x, T_i executes the following steps:

```
Get a shared lock on x
curVersion = maxV(x)
IF (curVersion > V(T_i))
    moveToFuture(T_i, u_i)
    V(T_i) = u_i
Read x(curVersion)
```

3. To write a data item y, T_i executes the following algorithm:

```
Obtain an exclusive lock on y
curVersion = maxV(y)
IF (curVersion > V(T_i))
    moveToFuture(T_i, u_i)
    V(T_i) = u_i
ELSE IF (curVersion < V(T_i))
        Create y in version V(T_i) by copying y(maxV(y)).
Update y(V(T_i)).
```

4. If subtransaction T_i creates child subtransactions, send these to their respective destination nodes.

5. If subtransaction T_i with no children completes, send a message $\mathsf{prepared}(V(T_i))$ to the parent subtransaction.

6. If subtransaction T_i with child subtransactions completes, wait to receive prepared messages from its children subtransactions.

7. When a subtransaction T_i with child subtransactions receives the $\mathsf{prepared}(v_j)$ messages from all its children, it computes a *global version of* T_i $\mathcal{V}(T_i)$, which is equal to the maximum of $V(T_i)$ and the version numbers received with the prepared messages from its children. If T_i is not root, it sends a $\mathsf{prepared}(\mathcal{V}(T_i))$ message to its parent subtransaction.

   If T_i is the root subtransaction, then $\mathcal{V}(T_i)$ is the *global version of the transaction*, $\mathcal{V}(T) = \mathcal{V}(T_i)$, and is the largest version number updated by any subtransaction of the transaction. The root subtransaction T_i decides whether to commit the transaction, and if so, sends a $\mathsf{commit}(\mathcal{V}(T))$ to itself and its children subtransactions.

8. When a subtransaction T_i receives the $\mathsf{commit}(\mathcal{V}(T))$ message, it checks whether or not its version $V(T_i)$ is equal to $\mathcal{V}(T)$. If $V(T_i) = \mathcal{V}(T)$, it means that T_i updated the *correct* version of data items, and the subtransaction can commit. As part of the commit process, T_i decrements $\mathsf{updateCount}(i, \mathsf{startV}(T_i))$, and sends a $\mathsf{commit}(\mathcal{V}(T))$ message to each of its children.

   If $V(T_i) < \mathcal{V}(T)$[1], it means that T_i updated an earlier version of data items than some other subtransactions of the same transaction T, and it now needs to move its updates into the later version $\mathcal{V}(T)$.

   At this time, two cases are possible: (1) $u_i = \mathcal{V}(T)$, meaning that version advancement has begun at node i, or (2) $u_i = V(T_i) = \mathcal{V}(T) - 1$, meaning that version advancement has not begun at node i. In the latter case, version advancement is initiated now, by treating the receipt of the $\mathsf{commit}(\mathcal{V}(T))$ message as a signal to start version advancement. (u_i is incremented and $\mathsf{updateCount}(i, \mathcal{V}(T))$ is initialized to 0.)

   Once it is ensured that version advancement is taking place ($u_i = \mathcal{V}(T)$), $\mathsf{moveToFuture}(T_i, \mathcal{V}(T))$ is executed to move subtransaction T_i into the global version $\mathcal{V}(T)$. Having done so, T_i commits, decrements $\mathsf{updateCount}(i, \mathsf{startV}(T_i))$, and sends a $\mathsf{commit}(\mathcal{V}(T))$ message to each of its children.

[1] $V(T_i)$ can be less than $\mathcal{V}(T)$ by at most 1

4 moveToFuture Function

Function moveToFuture(T_i, newV) switches an active update subtransaction T_i at node i from accessing version oldV $= V(T_i)$ to a new version newV. This function must bring the database on node i to a state the database would have if T_i never touched any data items in version oldV, and instead had operated all along with version newV.

The actions performed by moveToFuture depend on the log and recovery scheme used. If any of the *no-undo* (otherwise known as no-steal) schemes [BHG87] are employed, the only action required of moveToFuture is to update the current version number of the subtransaction: $V(T_i) =$ newV. Indeed, to avoid undoing any updates, these schemes apply updates to the actual database only at commit time; for active transactions, updates are kept in redo log records (for redo/no-undo schemes) or in temprorary *shadow* copies (for no-redo/no-undo). Since our protocol ensures that moveToFuture is only performed by uncommitted transactions, promoting transaction version numbers is sufficient to achieve the desired effect that transactions commit to their final version and have no effect on earlier versions. In addition, the final version number of T_i is included in the commit log record to ensure that, during recovery, T_i's log records are applied to the proper version.

In the recovery scheme of [BPR+96], active (non-committed) transactions are allowed to modify the database pages. However, the scheme ensures that, except during checkpoints, all modified pages as well as all undo log records of un-committed transactions remain in main memory. In this scheme, moveToFuture, while more complex than in the previous schemes, can still be implemented efficiently, without any disk accesses in most cases.[2]

In this case, moveToFuture(T_i, newV) reads log records from the tail of the log backwards, looking for records generated by T_i. For each redo record $redo(T_i, x)$, version newV of data item x is created by copying it from version oldV (recall that version oldV is still in main memory). For each undo record $undo(T_i, x)$, this record is applied to version oldV of x, thereby undoing the effect of T_i on this version. The function completes when it encounters the begin-transaction log record for T_i.

Note that moveToFuture achieves its goal by simply copying data items involved from the old to new version, not by aborting and restarting T_i. It is able to do that because each such data item has been locked by T_i by the time moveToFuture is invoked. In particular, it is guaranteed that none of these data items already exist in the new version due to another transaction. Thus, copying into the new version will not overwrite updates of other transactions.

The implementation of moveToFuture is similar for other redo/undo recovery schemes, such as Aries [MHL+92]. However, without special optimizations to logging along the lines of [BPR+96], moveToFuture within Aries is more likely to involve disk accesses. Therefore, in situations where version advancement is executed frequently, the scheme of [BPR+96] or no-undo schemes should be used.

5 Example Execution

Consider a distributed database with three sites i, j, and k, with data items w at i, x and y at j, and z at k. Initially, all the data is at version 0, and so are all the transaction counters. The current read version at each node is 0, and update version is 1. However, there is no data in version 1 as of yet.

[2]We can further reduce the likelihood of disk accesses during moveToFuture by coordinating checkpoint and version advancement operations, so that they are not done at the same time. Given that checkpoints of [BPR+96] do not interfere with user transactions, such coordination does not introduce any hidden synchronization between version advancement and user transactions.

TIME	SITE i	SITE j	SITE k
1	Update transaction T arrives; $\mathsf{startV}(T_i) = V(T_i) = 1$; $\mathsf{updateCount}(i, 1) = 1$.		
2	T locks w; creates and updates $w(1)$.		
3	T_j and T_k are issued to j and k.		Version advancement is initiated: $\mathsf{newu} = u_k + 1 = 2$,
4	Query R arrives; $\mathsf{queryCount}(i, 0) = 1$.	T_j arrives; $\mathsf{startV}(T_j) = V(T_j) = 1$; $\mathsf{updateCount}(j, 1) = 1$.	send *advance-u(2)* to i, j, k. Set $u_k = 2$,
5	R reads $w(0)$.	Update transaction S arrives; $\mathsf{startV}(S_j) = V(S_j) = 1$; $\mathsf{updateCount}(j, 1) = 2$.	send *ack-advance-u(2)* back to itself.
6	R completes; $\mathsf{queryCount}(i, 0) = 0$.		T_k arrives; $\mathsf{startV}(T_k) = (T_k) = 2$; $\mathsf{updateCount}(k, 2) = 1$.
7	Message *advance-u(2)* received, $u_i = 2$.	Message *advance-u(2)* received, $u_j = 2$.	T_k locks z; creates and updates $z(2)$.
8		Update transaction U arrives, $\mathsf{updateCount}(j, 2) = 1$.	T_k completes and sends Prepared(2) message to parent T.
9		U locks x; creates and updates $x(2)$.	
10		U commits, $\mathsf{updateCount}(j, 2) = 0$.	
11		T_j locks y; creates and updates $y(1)$.	
12		S attempts to lock y, and waits.	
13		T_j locks x and updates $x(2)$.	
14		T_j executes $\mathsf{moveToFuture}$, by copying y to version 2 and undoing its update on $y(1)$; $V(T_j) = 2$.	
15		Query Q arrives; $\mathsf{queryCount}(j, 0) = 1$.	
16		Q reads $x(0)$.	
17		T_j completes and sends Prepared(2) message to parent T.	
18	T receives Prepared messages from T_j and T_k; T executes $\mathsf{moveToFuture}$ by copying w to version 2 and undoing its change to $w(1)$.		
19	T commits and sends Commit(2) message to T_j and T_k; $\mathsf{updateCount}(i, 1) = 0$ Send *ack-advance-u(2)* to k.		
20		T_j receives Commit(2) and commits; $\mathsf{updateCount}(j, 1) = 1$.	T_k receives Commit(2) and commits; $\mathsf{updateCount}(k, 2) = 0$.
21		S locks y and updates $y(2)$. S performs a trivial $\mathsf{moveToFuture}$; $V(S_j) = 2$.	
22		S commits; $\mathsf{updateCount}(j, 1) = 0$. Send *ack-advance-u(2)* to k.	
23			All *ack-advance-u(2)* received. read version advancement starts: $\mathsf{newq} = q_k + 1 = 1$, send *advance-q(1)* to i, j, k.
24		*advance-q(1)* received; $q_j = 1$.	Set $q_k = 1$,
25		Query P arrives; $\mathsf{queryCount}(j, 1) = 1$.	Send *ack-advance-q(1)* to itself.
26	*advance-q(1)* received; $q_i = 1$.	P reads $y(1)$.	
27	Send *ack-advance-q(1)* to k.	P completes; $\mathsf{queryCount}(j, 1) = 0$.	
28		Q reads $y(0)$.	
29		Q completes; $\mathsf{queryCount}(j, 0) = 0$. Send *ack-advance-q(1)* to k.	
30			All *ack-advance-q(1)* received; Garbage collection of version 0 starts.

Table 1: Example Execution Sequence

Now consider a sequence of actions as shown in Table 4, involving update transactions S, T and U, and read-only queries P, Q and R. We use T_i to denote a subtransaction of transaction T at node i.

While not all the nuances can be captured in a single short example, there are a few interesting features of the execution in Table 4 that we would like to point out. Transaction T starts out in version 1 on node i and performs all updates on node i in version 1, but performs all updates on node k in version 2. On node j, it starts using version 1 but moves to version 2 as soon as it encounters data item x that exists in version 2 (step 13).

The existence of x in version 2 indicates that x was updated by a version 2 transaction that already committed (it committed because otherwise it would still hold the lock on x, preventing T from accessing x). This version 2 transaction conflicts with T, since both update x. Moreover, T cannot be serialized before this transaction because the version of x written by it does not reflect T's updates. Therefor, T must be serialized after this transaction, which it accomplishes by moving to version 2 and then updating x in version 2. Note that T's access to y in step 11 is in version 1 even though version advancement has begun on node j – at this point T has not yet developed a conflict with any version 2 transactions.

On node i, T never encounters a version 2 data, and therefore executes completely in version 1. This version mis-match is discovered at commit time, and T moves its updates to version 2 on i prior to committing.

Read transactions never obtain locks, and always see a consistent picture of the data. Read transaction P in step 26 reads version 1 of y, whereas read transaction Q only a few moments later (in step 28) reads version 0 of y.

6 Properties and Correctness of the AVA3 Protocol

6.1 Correctness of the AVA3 protocol

In this section, we sketch the proof of correctness of the AVA3 protocol.

6.1.1 Correctness of Version Advancement

The version advancement protocol never interferes with update and read transactions. However, to make progress it requires that values of update and query counters at each node go to zero, and that certain properties hold after these counters have gone to zero. We state the lemma for update counters; a similar lemma can be stated for the query counters.

Lemma 6.1 *Let each node maintain update counters for each version according to the* AVA3 *protocol, and let version advancement be done according to the* AVA3 *protocol start with* oldu *as the current update counter.*

Then, assuming that the messages to increment the update counter to newu *from the coordinator are eventually delivered to the target nodes, (1) Eventually* $\mathsf{updateCount}(i, \mathsf{oldu})$ *will be equal to* 0 *for all nodes i, (2) Once* $\mathsf{updateCount}(i, \mathsf{oldu}) = 0$ *is true, it will remain true at all points in the future (this is a stable property [HJPR87]), (3) Once* $\Sigma_i(\mathsf{updateCount}(i, \mathsf{oldu})) = 0$ *is true, the data for version* oldu *is consistent across all nodes, no update subtransaction with version number* oldu *is currently active at any node, and no update subtransaction with version number* oldu *will ever be active at any node.*

■

Proof: Since all new transactions are now assigned version number newu, it is guaranteed that data sets with any earlier version number will stop getting updated, with the update transaction counters becoming zero. If a subtransaction aborts, it will restart in version newu. If a node crashes, it will come back up with $\mathsf{updateCount}(i, \mathsf{oldu}) = \mathsf{updateCount}(i, \mathsf{newu}) = 0$, and any new subtransactions running on the node will increment $\mathsf{updateCount}(i, \mathsf{newu})$, and not increment $\mathsf{updateCount}(i, \mathsf{oldu})$. Once $\Sigma_i(\mathsf{updateCount}(i, \mathsf{oldu})) = 0$, it means that $\mathsf{updateCount}(i, \mathsf{oldu}) = 0$ at each node i, meaning that no update subtransaction is active on version i at any node. Consequently, the data in version oldu must be consistent across all the nodes. ■

6.1.2 Local Serializability:

When the version advancement process is not running, two phase locking guarantees that all update subtransactions at a node, acting on some version u_i, are locally serializable with each other. When version advancement is running, update subtransactions can act on two versions, oldu and newu, and the AVA3 protocol guarantees that these are still locally serializable with all update subtransactions with a commit version number oldu being serialized before all update subtransactions with a commit version number newu.

Lemma 6.2 *All query subtransactions with commit version v begin execution after all update transactions with commit version v complete execution.* ■

Proof: Read transactions are given a version number at the root subtransaction. Since update version numbers are always ahead of the query version numbers, the update subtransactions must have been started at nodes before the root subtransaction of the query transaction started. A version advance process must run between the update and a subsequent query transaction if the query transaction is to get the number v. Now the version advance process waits for all update transactions on version v to complete before advancing the update version and the read version numbers. A subsequent transaction that gets the read number v, clearly starts after the update transaction has completed. ■

Lemma 6.3 *Let version advancement processes, read subtransactions, and update subtransactions execute at a node according to the* AVA3 *protocol. All subtransactions at node i can be serialized as follows:*

- *All update subtransactions with commit version v are serialized according to the two phase locking protocol.*
- *All update subtransactions with commit version v are serializable after all update subtransactions with commit version $u < v$.*
- *All update subtransactions with commit version v are serializable after all query subtransactions with commit version $u < v$.*
- *All query subtransactions with commit version v are serializable after all update subtransactions with commit version v.*

■

Proof: The first two statements are straightforward.

For the third statement, we need to handle the case where a read subtransaction with commit version v needs to be serialized before any update subtransaction with commit version $v + 1$, even if the update subtransaction had a start version v and thus initially updated some data items in version v.

Consider an update subtransaction T_i and a read subtransaction Q_i executing on node i, and let T_i have commit version $v+1$ and start version v, and Q_i have commit version v. Let Q_i execute after T_i on this node. The query subtransaction Q_i can still be serialized before T_i. Indeed, one can show similar to the proof of Lemma reflemma:querysAfterUpdates that Q_i could start on node i only after T_i committed. Since T_i's start version is different from its commit version, it must have executed moveToFuture($T_i, v+1$) function before committing, and therefore before query Q_i started at this node. The moveToFuture($T_i, v+1$) function undoes all effects of update subtransaction T_i on version v, so that the query Q_i running later does not see any effect of T_i, and can be serialized before T_i.

The fourth statement follows from the Lemma 6.2, showing that if a read and an update transaction have the same version number, then the update transaction completed execution before the read transaction even began execution. ■

We thus have:

Theorem 6.1 *The* AVA3 *protocol produces locally serializable schedules at each node.* ■

6.1.3 Global Serializability

To show that the AVA3 protocol produces globally serializable schedules, we first note that:

Lemma 6.4 *Using the* AVA3 *protocol, all subtransactions of a transaction T have the same commit version number on all nodes.* ■

Proof: For read transactions, all subtransactions use the start version number of the root subtransaction as their commit number. For update transactions, if the subtransactions execute in different versions, the commitment protocol moves them up to the latest version that any subtransaction executes in. As a result, all update subtransactions T_i have the same commit version number. ■

Given that all the subtransactions with the same commit version number are locally serializable, the distributed two phase locking protocol guarantees that all transactions with the same commit version are globally serializable. Two transactions with different version numbers $u < v$ execute on versions u and v on all nodes. Hence, they have the same serialization relation on each node, and this must be their global serialization order.

We can thus derive the equivalent serial schedule from the transaction commit version numbers, and obtain our correctness result:

Theorem 6.2 *A schedule produced by the* AVA3 *protocol is equivalent to some serial schedule in which transactions are partially ordered by their version number, and within transactions of the same version number, the update transactions precede the read transactions.*

The AVA3 *protocol produces a globally serializable schedule of subtransactions.* ■

Proof: By Lemma 6.1, all subtransactions on a single node are locally serialized according to the properties of the serial schedule claimed in the current theorem. Further, by Lemma 6.4, all subtransactions of a transaction have the same commit version on each node. Thus the relative ordering between transactions of different versions is the same on all nodes. The relative ordering between queries of the same commit version is not relevant, as all orderings are equivalent. The ordering between update subtransactions of the same version number is guaranteed to be the same on all nodes by virtue of using a distributed two-phase locking protocol. Consequently, the claimed global serial schedule exists. ■

6.2 Performance Properties

It is easy to check that the AVA3 protocol described in Section 3 satisfies the following properties.

1. While the version advancement process is not running, (a) Only two versions of each data item exist, (b) The query version number q_i is the same at all nodes, (c) The update version number u_i is the same at all nodes, and (d) $u_i = q_i + 1$.

2. While the version advancement process is running, (a) At most three versions of each data item exist, (b) For every pair of nodes i and j, if $u_i \neq u_j$, then $q_i = q_j$, and (c) For every pair of nodes i and j, if $q_i \neq q_j$, then $u_i = u_j$.

3. For every node i, the condition $q_i < u_i \leq q_i + 2$ holds at all times.

4. An update subtransaction T_i with a start version number v may read and update data in version $v + 1$. If T_i reads or updates data in version $v + 1$, its commit version number is $v + 1$. Otherwise, T_i's commit version number is v.

We now show that we have achieved our goal of global serializability without any synchronization between version advancement, queries, and update transactions. We note that all updates to counters are increments or decrements, and all updates to version numbers are increments. Increment/decrement operations do not conflict, and only require latches before being executed, thereby giving us the desired non-delaying, non-blocking, and non-starvation properties of the read, update, and version advancement operations, respectively.

Theorem 6.3 *Using the* AVA3 *protocol, modulo latches on counters and version numbers, (a) read subtransactions are delay-free (they cannot be delayed by any other queries, updates, or version advancement), (b) updates subtransactions are block-free with respect to read transactions and version advancement. (The only possible delay to an update subtransaction due to version advancement is the time to execute a* moveToFuture *function.), and (c) version advancement is starvation-free (it cannot be delayed indefinitely by newly arrived transactions).* ■

Proof: The read subtransactions obtain no locks, and only need to modify counters. Hence, they are not delayed by other queries, updates, or version advancement. Update subtransactions do obtain locks using a distributed two-phase locking protocol, but doing so only causes blocking between update subtransactions, and not between update and read subtransactions, or between update subtransactions and version advancement. Note that version advancement can cause an update subtransactions to execute the moveToFuture function, perhaps delaying the update, but this does not block the update. After version advancement starts, Lemma 6.1 states that eventually $\mathsf{updateCount}(i, \mathsf{oldu})$ will be equal to 0 for all nodes i, allowing the system to shift to a new update version. It can similarly be shown that eventually $\mathsf{queryCount}(i, \mathsf{oldq})$ will be equal to 0 for all nodes i, allowing the system to shift to a new query version, and thereby ensuring that version advancement will complete. ■

7 AVA3 Protocol in a Centralized System

Our protocol becomes much simpler for a centralized system. Version advancement still goes through the same multiple phases, but only at the one node: increment the update version counter, wait until no update transactions are operating on the old update version, increment the read version counter,

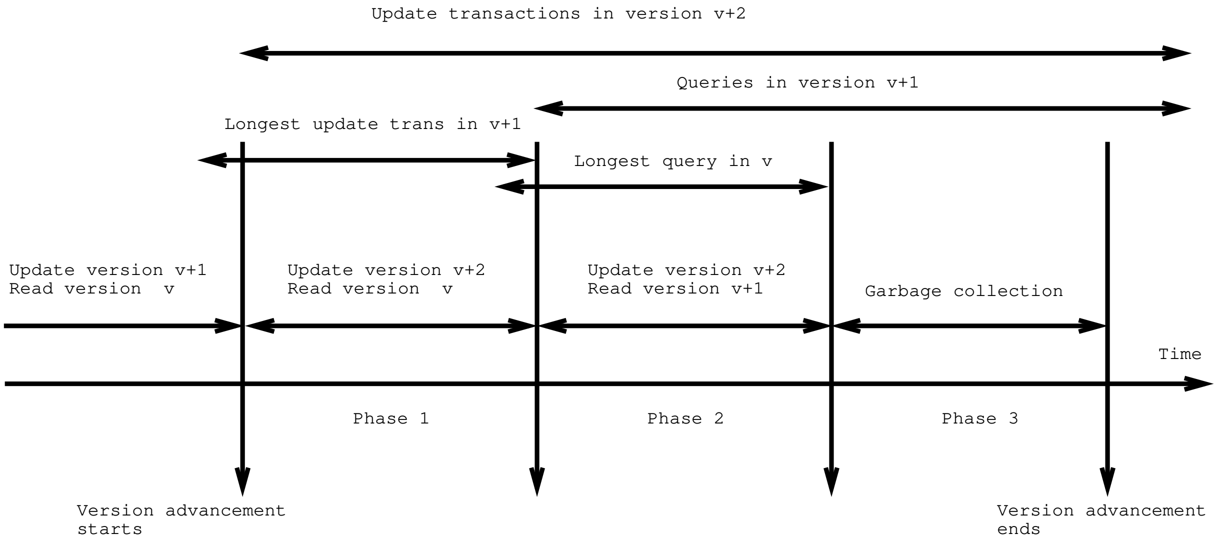

Figure 1: Time diagram of version advancement.

wait until no query transactions are reading the old read version, and finally garbage collect the old versions.

Read transactions do nothing special. Update transactions operate as in the distributed case, copying to the current version whenever they update a data item for which the latest version is outdated, and applying the moveToFuture function whenever they encounter a data item with a version later than their own. However, there is no distributed commitment, so an update transaction can simply commit when it completes.

Thus we have a simple centralized protocol that performs asynchronous version advancement and guarantees that no more than 3 versions of a data item are ever created at a time.

8 Limiting Data Staleness

The AVA3 protocol works by running periodic version advancements to make more recent data available to queries. However, the way we described the protocol, the queryable data may never be absolutely up-to-date, as the following discussion illustrates. In other words, while version advancements make increasingly fresher data available for reads, this data may never catch up with the newest data.

Assume for simplicity that message delivery times are negligible compared to the transaction execution times. Consider a timing diagram of version advancement shown in Figure 1, where initially the read version number is v and update version number is $v+1$ on all nodes. Then, after the version advancement begins, it makes data in version $v+1$ available for reads after the longest update transaction, which was active when this version advancement started, completes. During this time, new update transactions, by creating and modifying data in version $v+2$ can make data in version $v+1$ obsolete. Thus, when data in version $v+1$ becomes available for reads, it may be already behind the newest data. The lag, however, is bounded by the duration of the longest update transaction that was active at the time the version advancement started.

We can drastically reduce this lag using the following simple optimization of the AVA3 protocol. When an update subtransaction executes the moveToFuture function at a site i, it is possible for it to decrement the update counter for its start version number, and increment the update counter for its new current version number immediately after rolling back updates in the old version. For the purpose of version advancement, the update transaction then appears to have "started" in the advanced version at the node. Thus, Phase 1 of the protocol can complete without waiting for a long-running update transaction that started in the old version to finish. The duration of Phase 1, which determines the maximum lag above, becomes limited to the time needed by all transactions that were active when version advancement started to execute moveToFuture.

As already mentioned, with appropriate optimizations of recovery scheme employed, moveToFuture is either a virtual no-op (if a no-undo recovery scheme is used) or it can be done entirely in main memory and therefore be fast. This reduces the maximum lag of newly written data for reads. In other words, if a user decides it needs the freshest data for a query, and no version advancement is in progress, the user can start a new version advancement immediately prior to the query, and the data used by the query will be (almost) current.

Another interesting question is as follows: If we run version advancements as frequently as possible, what is the upper bound on the staleness of data being read. The AVA3 protocol stipulates that version advancements to different newu cannot overlap. We can actually relax the non-overlapping restriction and permit the garbage collection phase to complete in the background, requiring only that the first two phases of version advancement do not overlap.[3] Consider Figure 1 again, except assume that the duration of Phase 1 is negligible due to the optimization discussed above. Assume that, immediately after version advancement to update version $v+2$ completes Phase 2, version advancement to update version $v+3$ starts. During Phase 2 of version advancement to update version $v+2$, queries read increasingly obsolete data in version $v+1$, due to concurrent update transactions executing in version $v+2$. Once version advancement to update version $v+3$ starts, it makes current data of version $v+2$ available for new queries. Since the duration of Phase 2 is bounded by the longest existing query, we conclude that in this mode, a new query Q operates on a a snapshot of the database that may be obsolete by at most the age of the longest query which was executing at the time Q started.

9 Related Work

There has been much work to avoid interference between read-only and update transactions by creating multiple versions of data on updates. Such schemes have even been implemented in products, going back to Prime Computer's CODASYL system and Digital's Rdb/VMS.

The algorithms of [BHR80, SR81] never keep more than two versions of data, compared to our three versions. However, in [BHR80], a read-only query may delay the commitment of an update transaction, an update transaction may be aborted because of the versioning mechanism, and read-only queries incur overhead of obtaining read locks. The algorithm of [SR81] may, in addition, also delay or abort queries. The algorithm of [Wei87] require that read-only transactions write meta-information in the data items they read, and that update transactions write meta-information in all accessed data items at commit time. Thus, these techniques do not satisfy our non-interference requirements.

Several algorithms have been proposed that, like ours, satisfy the non-interference requirements. Algorithms of [CFL+82, AS89, WYC91, BC92, MPL92] address the centralized database,[4] while

[3] While this may increase the number of copies of the data due to slow garbage collection, still only the latest three copies are used by user transactions.

[4] While [MPL92] contains the extension to distributed case, version advancement in the distributed case can interfere

[CG85] extends [CFL+82] for the distributed case. Techniques suggested in [CFL+82, CG85, AS89, BC92] achieve the non-interference requirement at the cost of potentially unlimited number of versions, while our AVA3 protocol creates at most three versions. On the other hand, a read transaction in these protocols always reads the latest version written by an update transaction with an earlier timestamp. In our AVA3 protocol, read transactions access increasingly obsolete data until the next version advancement occurs.

Similar to AVA3 , protocols of [Wei87, CG85] modify the commit protocol of update transactions so that participating sites piggyback some information on their prepare messages. In [Wei87], this information includes timestamps of all accessed data items and is used to let the transaction coordinator choose a unique timestamp that is greater than all those timestamps. This new timestamp is then sent to participants in the commit message and assigned to the accessed objects. In [CG85], prepare messages carry local *committed transaction lists*, which are lists of transactions known to the participants to have committed. They are used by the coordinator to compute and distribute to the participants a new list, that includes transactions in all those lists. This is the way for the participants to learn about transactions committed by other sites.

In our protocol, prepare messages carry the version numbers used by subtransactions, and are used to detect any possible version mis-match. The mis-match is detected by each participant individually by comparing the version it sent with prepare message with the version received with commit message, which is the greatest version reported by any participant. Upon detecting a mis-match, a participant moves the subtransaction into the version received with the commit message by means of moveToFuture function, and then commits.

[GO96] proposes an interesting scheme to use old versions cached at the clients as different version of the data. Transactions choose *slices* in which to execute. A slice is a mutually consistent subset of pages in the database. The server maintains the information which pages belong to which slices, and a directed conflict graph that orders slices according to the serialization order of transactions executed in those slices. When a transaction needs a page that does not currently belong to its slice, the server expands the slice unless it creates a cycle in the slice graph, in which case the transaction is aborted. Generalizing this scheme to a distributed database appears difficult because all servers would have to have a consistent global view on the slice conflict graph.

[AS89] separates concerns of version control and concurrency control, by implementing the versioning mechanism in such a way that it can be used in conjunction with any concurrency control algorithm. Our protocol is predicated on the use of two-phase locking. Whether our idea can be used in conjunction with other concurrency control protocols remains a question for future work.

For the centralized case, the protocols of [WYC91, MPL92] requires up to four versions of data to guarantee non-interference of version advancement and user transactions, and the algorithm of [BC92] can be configured to require at most four versions (with the view sharing idea of Section 2.3 in [BC92]). Our AVA3 protocol in the centralized case reduces to a scheme that needs at most three versions to guarantee non-interference. On the other hand, in these protocols, read transactions that start immediately after version advancement always access the latest committed data, and then start falling behind. In our protocol, while version advancement makes more recent data available for read transactions, they still may access out-of-date data even right after version advancement. Since read transactions are allowed to access out-of-date data anyway, this seems to be a small penalty for reducing the number of versions from four to three.

Finally, the protocol of [JMR97] targets the case when update transactions commute on individual nodes. For this workload, the protocol creates at most three versions of data and avoids *any*

---

with and cause aborts of user transactions.

synchronization delays. Our AVA3 protocol can also be applied to commuting updates, though it will lead to some updates becoming available for reads later than in the [JMR97] protocol.

10 Conclusions

The AVA3 protocol presented in this paper was motivated by several large distributed database applications within AT&T. These applications have continuous update streams and queries that need access to data. The volume of both updates and queries is so high that the performance of these systems takes precedence (up to a point) over the freshness of the data returned by queries. However, while queries do not require the very latest data, returning consistent results is highly desirable.

These requirements are summarized by our non-interference requirement, which stipulates that (1) read-only queries be completely decoupled from updates, (2) either read-only queries or update transactions be never delayed on the account of version management, and (3) read-only queries be not required to acquire any locks or write any control information to data items they access.

As database sizes continue to grow, these requirements will become increasingly common. Indeed, the usual manual versioning that exploits periodic flushing of updates to the read copy will involve ever-longer disruption periods, when user operations are prohibited. On the other hand, existing distributed multiversioning schemes that satisfy the non-interference requirement may be expensive in a large database due to potentially unlimited number of versions they can create. Not only they consume disk space, but managing versions becomes more complicated in this case.

The AVA3 protocol satisfies the non-interference requirement while creating at most three versions. At the same time, the staleness of data returned by queries can be effectively controlled by the frequency of version advancement. In the limit, with sufficiently frequent version advancement, a query always receives a data snapshot that is never older than the age of the longest concurrently active query. Moving beyond this bound may require additional versions.

Several optimizations are possible to the AVA3 protocol described here. For example, an update transaction can carry a version number with it, and use the maximum of this and the current update version, u_i, at a node i, to determine its current update version at site i. By so doing, the number of moves to the future can be decreased. Secondly, for read transactions it suffices if the root subtransaction increments (and later decrements) a transaction counter: there is no need for the child subtransactions to do so. Thirdly, it is the case that read transactions operate on a version only after all updates are done with the version, and version advancement has moved the read version forward. Therefore, the same transaction counter can be used for both read and update, rather than having separate update and query transaction counters for each version at each node.

Finally, our protocol is extremely simple. In addition to usual data structures for maintaining multiple versions (which are also simplified due to the fixed number of versions), it only maintains a few integers per node and two integers per transaction. We therefore hope that this protocol will find practical usage in real systems.

References

[AS89] D. Agrawal and S. Sengupta. Modular synchronization in multiversion databases: Version control and concurrency control. *ACM SIGMOD Conf. on the Management of Data 89, (Portland OR), -Jun..*, May 1989.

[BC92] P. Bober and M. Carey. On mixing queries and transactions via multiversion locking. In *Proc.IEEE CS Intl.Conf. on Data Engineering 8, Tempe, AZ.*, February 1992.

[BHG87] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[BHR80] R. Bayer, H. Heller, and A. Reiser. Parallelism and recovery in database systems. *ACM Trans. on Database Systems*, 5(2):139–156, June 1980.

[BPR+96] Phillip Bohannon, J. Parker, Rajeev Rastogi, S. Seshadri, Avi Silberschatz, and S. Sudarshan. Distributed multi-level recovery in main memory databases. In *Proceedings of the Fourth International Conference on Parallel and Distributed Information Systems (PDIS)*, Miami Beach, FL, December 18-20 1996.

[CFL+82] A. Chan, S. Fox, W-T.K. Lin, A. Nori, and D.R. Ries. The implementation of an integrated concurrency control and recovery scheme. In *ACM SIGMOD Conf. on the Management of Data 82, Orlando FL.*, pages 184–191, June 1982.

[CG85] A. Chan and R. Gray. Implementing distributed read-only transactions. *IEEE Transactions on Software Engineering.*, 11(2):205–212, February 1985.

[GO96] S. Gukal and E. Omiecinski. Transient versioning for consistency and concurrency in client-serer systems. In *Proc. 4th IEEE Int. Conf on Parallel and Distributed Information Systems*, pages 274–285, 1996.

[HJPR87] J. M. Helary, C. Jard, N. Plouzeau, and M. Raynal. Detection of stable properties in distributed applications. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, pages 125–136, 1987.

[JMR97] H. V. Jagadish, I. S. Mumick, and M. Rabinovich. Scalable versioning in distributed databases with commuting updates. In *IEEE International Conference on Data Engineering, to appear*, 1997.

[MHL+92] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwartz. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS)*, 17(1):94–162, March 1992.

[MLO86] C. Mohan, B Lindsay, and R. Obermarck. Transaction management in the R* distributed database management system. *ACM Transactions on Database Systems*, 11(4):378–396, 1986.

[MPL92] C. Mohan, H. Pirahesh, and R. Lorte. Efficient and flexible methods for transient versioning of records to avoid locking by read-only transactions. In *ACM SIGMOD Conf. on the Management of Data 92, San Diego.*, June 1992.

[SR81] R.E. Stearns and D.J. Rosenkrantz. Distributed database concurrency controls using before-values. In *ACM SIGMOD Conf. on the Management of Data 81, Ann Arbor.*, April 1981.

[Wei87] William E. Weihl. Distributed version management for read-only actions. *IEEE Transactions on Software Engineering (SE), Vol.SE-13 .*, SE-13(1), January 1987.

[WYC91] K. L. Wu, Philip S. Yu, and M. S. Chen. Dynamic, finite versioning for concurrent transaction and query processing. Technical Report RC 16633, IBM T. J. Watson Research Center, 1991.