Scalable Versioning in Distributed Databases with Commuting Updates

H. V. Jagadish
AT&T Laboratories
jag@research.att.com
http://www.research.att.com/~jag

Inderpal Singh Mumick
AT&T Laboratories
mumick@research.att.com
http://www.research.att.com/~mumick

Michael Rabinovich
AT&T Laboratories
misha@research.att.com
http://www.research.att.com/~misha

Abstract

We present a multiversioning scheme for a distributed system with the workload consisting of read-only transactions and update transactions, (most of) which commute on individual nodes. The scheme introduces a version advancement protocol that is completely asynchronous with user transactions, thus allowing the system to scale to very high transaction rates and frequent version advancements. Moreover, the scheme never creates more than three copies of a data item.

Combined with existing techniques to avoid global concurrency control for commuting transactions that execute in a particular version, our multiversioning scheme results in a protocol where no user transaction on a node can be delayed by any *activity (either version advancement or another transaction) occurring on another node. Non-commuting transactions are gracefully handled.*

Our technique is of particular value to distributed recording systems where guaranteeing global serializability is often desirable, but rarely used because of the high performance cost of running distributed transactions. Examples include calls on a telephone network, inventory management in a "point-of-sale" system, operations monitoring systems in automated factories, and medical information management systems.

1. Introduction

Distributed retailing and billing systems commonly fragment data amongst several databases, and have a high transaction rate so that running global concurrency control is impractical. At the same time, not enforcing global consistency can result in inaccurate information being given to customers, and incorrect results for audits and bookkeeping. We present a low overhead global consistency scheme for applications where update and read transactions satisfy certain commuting properties. Our work was motivated by a proprietary telephone billing application. An analogous hospital application is described below.

Motivating Example: Consider a large hospital with multiple departments and external providers, each maintaining its own accounting system and computing charges for its services independently. A visit by a patient results in charges from several departments. Moreover, at the time a physician orders procedures, collects samples for tests, etc., the final charge amount for different procedures is typically not known to the physician or to the customer.

The recording of a patient visit is thus a multi-database update transaction that updates databases of several departments and records the charges incurred in these departments. One such update transaction may be represented as $T1 = \{w_{11}(x_1), w_{12}(x_2)\}$, where $w_{11}(x_1)$ represents a write into a patient's record in the radiology department's database (record the procedure done and charge applied, increment total charge due), and $w_{12}(x_2)$ represents a similar write into the record of the same patient in the pediatric department's database. We use the subscript ij to denote an operation by transaction T_i at database node j. There are also simultaneous read operations in response to patient inquiries, and to generate billing statements. One such read transaction may be represented as $T2 = \{r_{21}(x_1), r_{22}(x_2)\}$, where $r_{21}(x_1)$ represents a read of the patient's total charges in the radiology department's database, and $r_{22}(x_2)$ represents a read of the same patient's total charges in the pediatric department's database. Transactions $T1$ and $T2$, and others like them, are meant to run concurrently, as illustrated in Figure 1.

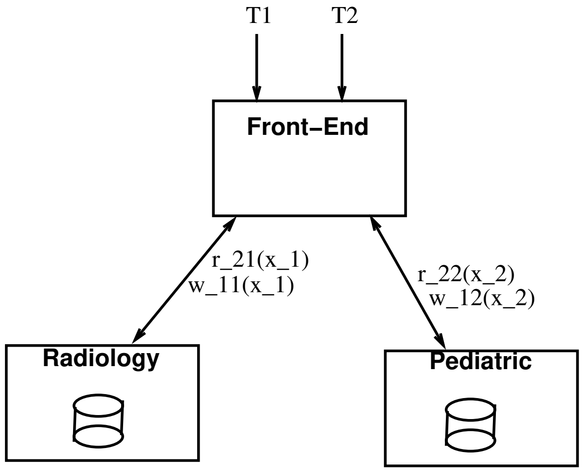

Figure 1. Transactions in a distributed hospital system.

There are four options on how these transactions might be handled by the system.

- Global Synchronization: The system can treat all global transactions (visit, billing, and inquiry transactions in the hospital example) as full-fledged distributed transactions, performing global concurrency control and two-phase commitment. This solution guarantees *global serializability* of transactions [4], where each global transaction takes effect on either all or none of the nodes involved, and all transactions appear to execute in a sequential order. However, the delays due to global synchronization are often prohibitive. For our example, it means that a schedule $\{w_{11}(x_1), r_{21}(x_1), w_{12}(x_2), r_{22}(x_2)\}$ will not be allowed, and the read transaction will have to wait for the update transaction to complete.

- No Coordination: Global transactions can run without global synchronization between nodes. This way, there is no performance loss due to coordination, but correctness is sacrificed. For our example, the schedule

  $$\{w_{11}(x_1), r_{21}(x_1), r_{22}(x_2), w_{12}(x_2)\}$$

  will be allowed, and a patient enquiring about his balance due will see only partial charges from procedures performed during a single visit, and this is an undesirable situation. (The analogous situation in a telephone billing system is that a customer sees only partial charges from a call on a bill.)

- Manual Versioning: One can accumulate update transactions for some period, say a month, in a new version that is not available for reading. This is the typical way that companies try to avoid the problems with the two alternatives above. Some time after the month ends, we hope that all updates have been applied to that month's version so that these values are stable and can be made available for read queries. Meanwhile, accumulation of update transactions for the next month takes place in a new version. This scheme is designed for batch processing, and the read operations are always behind by up to a month (or other chosen period).

  Whether such a scheme is acceptable depends on how up-to-date we need the accessed information to be in the application at hand. For our example, the write transaction would increment the balance due in the current month, while the read transaction would read the balance due from last month. Thus, a customer making an enquiry will not see any charges from the current month's bill. Further, if the transaction $T1$ were to start execution on January 31, one or both of the writes may be delayed beyond the version switchover date, and a bill generation query on February 10 may not report the charges from the January 31st procedures, or worse, may still report only a part of the charges from the January 31st procedures. In other words, correctness is not guaranteed. To reduce the possibility of incorrect executions, the delay between the time when update transactions start accumulating in a new version and when read transactions are allowed to use the next version is usually set conservatively high. This introduces additional (and often unnecessary) staleness of data available for reads. Finally, this scheme involves rigid administrative procedures and lacks flexibility in choosing version periods.

- Desired Solution: We would like to automate the process of maintaining versions against which read and update operations occur, and advance versions as soon as deemed necessary so that read operations can access more current data. 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 different versions exceeds some threshold, or after a particular update transaction commits. Further, we would like to ensure that read transactions like $T2$ always either read all the charges due to a hospital visit, or read none of the charges due to a hospital visit. In other words, we would like to *guarantee* correctness and avoid long delays in making new data available for reading. Moreover, for scalability, we want to ensure that (1) read and update operations do not have to wait for co-ordination between nodes, and (2) read and update transactions are not blocked on account of version advancement.

The class of applications of greatest interest to us, exemplified by the hospital billing situation described above, is characterized by the property that update transactions commute with other update transactions at each component database, while *update transactions do not commute with read transactions*. This class of applications, called data recording systems, is further discussed in Section 6.

By exploiting the commuting property, it is easy to achieve global serializability without ever delaying a user transaction on a node n due to any read or update transactions occurring on nodes other then node n through the use of multiversioning, as follows: keep two versions of the data, make all read transactions access one (older) version, and all update transactions access the other (current) version. Since all reads commute and all updates commute, and since updates and reads access distinct versions, we are (almost) done. The cost we pay is that the read-only transactions may return an older version of the data. (In fact, this is exactly what the manual versioning scheme described above would suggest). To bring the older version up-to-date, a *version advancement* process is required that makes the current update version into the new read version and creates a new current update version. However, a naive version advancement technique requires global synchronization between version advancement and the read and update transactions on all nodes. Such a version advancement would then slow the read and update transactions, and cannot be run too frequently.

Contributions: The main contribution of this paper is a multi-version protocol (the 3V algorithm) with version advancement that is completely asynchronous with user transactions. We also show that at most three versions of the data are needed.

The absence of global synchronization does not mean that there is no communication between nodes in the distributed system. However, messages exchanged in our algorithm are sent asynchronously with respect to the execution of user transactions, and introduce no waiting in the execution or commitment of user transactions.

The 3V algorithm allows for a graceful introduction of non-commuting updates.

Paper Outline We begin with an informal description of the 3V algorithm in Section 2, and then introduce the transaction model in Section 3. The 3V algorithm is presented in Section 4. An extension to handle non-commuting updates is described in Sections 5. Section 6 describes the application domain of data recording systems where the 3V algorithm can be applied. Related work is discussed Section 7, and we conclude with Section 8.

2. Outline of the Versioning Algorithm

Before presenting our formal model and algorithm in the next sections, we informally explain the central idea here.

2.1. The Basic System

We consider a distributed database, where a transaction can span multiple nodes. Portions of a transaction executed on individual nodes are referred to as *subtransactions*. We assume a tree model of transactions [16], where a transaction is first submitted to one server, which performs its subtransaction and then sends subtransactions down to other servers for further execution. These servers may in turn send more subtransactions to other servers, possibly causing the transaction to visit some servers multiple times. A transaction, therefore, consists of one or more partially ordered subtransactions, with one *root* subtransaction preceding all others. (In Figure 1, the empty subtransaction in the front-end system functions as the root subtransaction.)

Conceptually, our approach assumes that all nodes initially have two versions of the data, version 0 and version 1. Updates are performed against version 1 of the data, while read-only queries use version 0. (In fact, a data item of version 1 is created only when an update transaction actually writes it. When an update transaction needs to read a data item that does not yet exist in version 1, it reads version 0 of this item.)

Since we assume that update subtransactions commute, any local serialization of the update subtransactions leads to a global serialization of the update transactions. The read transactions can be brought into the global serialization by placing them before all update transactions.

If this were all to it, the data used by read transactions would become increasingly obsolete. To prevent reads from accessing increasingly obsolete data, we introduce a version advancement process. The idea is to create a third copy of the database (with version number 2) on every node, instruct all nodes to start executing new update transactions against this new copy, wait until all old update transactions, which have been running against version 1, complete execution (making version 1 copy of the database consistent across all servers), and instruct all nodes to use version 1 for future read-only queries. Version 0 of the database can then be discarded as soon as all queries using it terminate.

The challenge that we address is to make sure that the version advancement activity, including detection of termination of updates on version 1, does not interfere with the user transactions. In other words, we want that there be no global synchronization between version advancement and user transactions.

2.2. Version Advancement

The version advancement algorithm begins by notifying all nodes that they should start a version advancement process. Receipt of this notification is not synchronized between nodes. After a node receives the notification to move to a new version, it executes any new root update subtransactions submitted to the node against a third copy of the database, which is given version 2. The data in version 2 is created only when a root update subtransaction is assigned version 2 and writes data onto this version. Thus, it is possible that a subtransaction executing in version 2 needs to read a data item x in version 2, but the data item x was never created in version 2. In such a case, the subtransaction reads the latest existing version of the data. The above mechanism of "copying on update" saves space, and avoids the need for creating a consistent snapshot of the database, which would have interfered with user update transactions.

We must ensure that all subtransactions of a transaction execute against the same version of data, in spite of the asynchronous nature of the version advancement. This is accomplished by the root subtransaction associating a version-id for the entire transaction, which is then carried by each of its descendant subtransactions. When a descendant subtransaction arrives at its target node that has not yet been notified of the version advancement, the target node infers, from the version-id of the arriving subtransaction, that version advancement has been initiated and treats the arrival of this subtransaction as the notification. On the other hand if a descendent update subtransaction that is supposed to execute on an older version arrives at a node that has already initiated version advancement to a new version, we execute the subtransaction against *both* copies of the data, version 1 and version 2.

After it is established that all nodes have been notified (asynchronously) about the initiation of a version advancement, and thus all new update transactions will run against data version 2, the version advancement algorithm needs to wait until all update transactions still running against version 1 are complete. It is not sufficient just to check on every node whether there is any transaction running on version 1: a subtransaction running on version 1 on node p might have sent a child subtransaction to node q and committed on node p; while the child subtransaction is in transit, no server may be running any transactions against version 1. However, it is not the case that update transactions against version 1 are complete.

We propose a simple and efficient way to establish that all transactions executing against a particular version have completed. We associate two counters with every version for every pair of nodes, a completion counter (C) that counts the number of completed subtransactions, and a request counter (R) that counts the number of subtransaction requests sent from one node to the other. When a root subtransaction arrives at node p and is assigned a version-id v, it increments the request counter R_{vpp} (request from node p to node p for executing a subtransaction on version v). When a subtransaction running on node p against version v sends a child subtransaction to a node q, it increments the request counter R_{vpq} (request from node p to node q for executing a subtransaction on version v). When the root subtransaction running on node p against version v terminates, it increments completion counter C_{vpp} (completed a subtransaction from node p to node p on version v). When a subtransaction running on node q invoked from node p against version v terminates, it increments completion counter C_{vpq} (completed a subtransaction from node p to node q on version v). All transactions against version v have completed when each request counter R_{vpq} has a value equal to completion counter C_{vpq}, and we are guaranteed that no new root subtransactions will run on version v.

To preserve locality, request counters R_{vpq} are located at node p, and completion counters C_{vpq} are located at node q. Checking for their equality would appear to require a global transaction that locks these counters and reads them. However, following ideas suggested for checking of stable properties in the distributed computing literature [8, 12, 9], we can show that asynchronous reading of these counters also guarantees a correct result, thereby avoiding the need for any global synchronization between version advancement and user transactions.

2.3. Example Execution

Consider a distributed database with three sites p, q, and s, with data items A and B at p, D and E at q, and F at s (Figure 2). Initially, all the data is at version 0, and so are all the counters. However, the "current" update version for each data item is version 1. (Version 0 is read only). Now consider a sequence of actions as shown in Table 2.3. The TIME column shows the real time of event occurrence - no assuption is made of the existence of a global clock in the system.

While not all the nuances can be captured in a single short example, there are a few interesting features of the execution in Table 2.3 that we would like to point out. Transaction j executing against version 2 at site q spawns a subtransaction j_p (time 12) that arrives at node p (time 19) before it has been notified about version advancement. Obviously, this descendant cannot execute against version 1 at p. Otherwise, version 1 of the database would remain inconsistent across all nodes even after all update activity against version 1 terminates. To deal with this problem, every descendant subtransaction of a root j carries the version number against which j executed. When a descendant sub-

TIME	SITE p	SITE q	SITE s
1	Update tx i arrives at node p $R_{1pp} = 1$		
2	Tx i updates version 1 of data item A		
3	Subtx i_q issued to site q $R_{1pq} = 1$		
4	Subtx i_s issued to site s $R_{1ps} = 1$		
5	Read tx x arrives at node p		
6	x reads A version 0		
7			i_s arrives
8			i_s updates F version 1 $C_{1ps} = 1$
9		Version advancement begins	
10		Update tx j arrives $R_{2qq} = 1$	
11		j updates D version 2	
12		j_p subtx issued to site p $R_{2qp} = 1$	
13		Subtx i_q arrives	
14		i_q updates D versions 1 and 2	
15		i_q updates E version 1	
16		i_{q_p} subtx issued to site p $R_{1qp} = 1$	
17		Read Tx y arrives	
18		y reads D version 0	
19	Subtx j_p arrives from node q, has new version 2; Node p begins version advancement Node p advances update version to 2		
20	Subtx j_p updates version 2 of data item A $C_{2qp} = 1$		
21	Version adv. notice (for version 2) arrives Update version already advanced to 2		
22		j_p completion notice arrives $C_{2qq} = 1$ j is complete	
23			Version adv. notice arrives
24	Subtx i_{q_p} arrives from node q		
25	Subtx i_{q_p} updates version 1 of data item B $C_{1qp} = 1$		
26		i_{q_p} completion notice arrives $C_{1pq} = 1$	
27	Completion notice for subtx i_q arrives from node q		
28	Completion notice for subtx i_s arrives from node s i is complete $C_{1pp} = 1$		

Beyond this point all version data values are stable, all version counters match up. A coordinator can determine this by means of an asynchronous read of the counters, and then inform each site, asynchronously of a read version advancement. Until each site is notified to advance its read version, all read transactions will continue to go to version 0.

Table 1. Example Execution Sequence

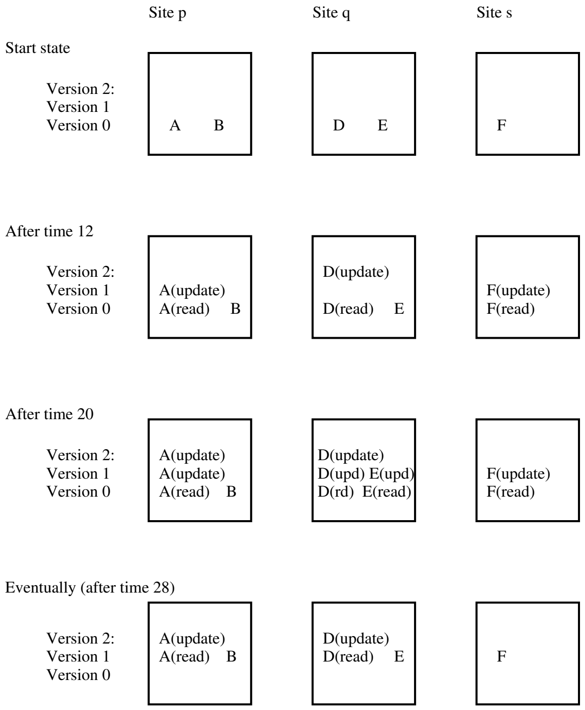

Figure 2. Example Scenario

transaction arrives at a node that has not yet been notified of the version advancement, the node infers that the version advancement has been initiated and treats the arrival of this subtransaction as the notification. Thus, this subtransaction (as well as all new root subtransactions at this node) will run against version 2.

A complementary situation occurs when the root transaction i, still executing against version 1 on node p, spawns a subtransaction i_q that arrives at a node q (time 13) after it already switched to version 2 (at time 9). i_q cannot execute against version 1 on q. Otherwise, version 2 of the database on this node would not reflect the result of i_q. Yet when p creates version 2 of its data, it will reflect the result of i. Thus, version 2 of the data will always be inconsistent on p and q: it will reflect the root subtransaction of i on node p but not i_q on node q. On the other hand, i_q cannot run against version 2 on q either, since this would leave version 1 inconsistent, reflecting i on p but not i_q on q.

To resolve this dilemma, we perform i_q on q against *both* copies of the data item D, version 1 and version 2 (time 14). However, data item E does not yet have a version 2 copy at site q at time 15. Therefore i_q needs to execute only against version 1. In other words, the overhead of performing two updates instead of one applies only when there is data contention that would, in an ordinary system, have blocked the transaction from performing any update at all.

3. The Model and Terminology

3.1. Commutativity

Two operations commute if the computational effect of their execution is independent of the order in which they are processed [4]. Two transactions (or subtransactions) are said to commute if their relative order of execution is immaterial to the value returned by each as well as to the final state of the database. Clearly, two subtransactions $S1$ and $S2$ commute if every operation in $S1$ commutes with every operation in $S2$. However, we note that subtransactions can commute even if their operations do not commute.

EXAMPLE 3.1 Consider the subtransactions $S1$ and $S2$ below:

$$S1 = \{read(x); x = x{+}5; write(x); read(y); y = y{+}5; write(y)\}$$

$$S2 = \{read(x); read(y); z = x - y; write(z); return(z)\}$$

The individual operations in $S1$ and $S2$ do not commute; however $S1$ and $S2$ commute, since the returned value and database state due to $S1$ after $S2$ is the same as the returned value and database state due to $S2$ after $S1$. □

In this paper, we require that update subtransactions commute, as captured by the following definition. We do not require the individual operations to commute. We assume that a local concurrency scheme serializes update subtransactions on each node.

Definition 3.1 Well-behaved set of transactions: A set of transactions $\mathcal{T} = \{T_1, \ldots, T_m\}$ is well-behaved if each pair of subtransactions (S_{i_k}, S_{j_l}) in any two transactions T_i and T_j (where i may be equal to j) are commuting. Transactions that are members of this set are *well-behaved* transactions with respect to set $\mathcal{T}$. □

The transactions in our system will be partitioned into two sets – the *read* set, $\mathcal{R}$, comprising read-only transactions, and the *update* set, $\mathcal{U}$, comprising transactions with at least one subtransaction that includes a write operation.

The read set, $\mathcal{R}$, is well-behaved by definition, and we assume, except in Section 5, that the update set, $\mathcal{U}$ is also a well-behaved set of transactions. However, the union of the read and update sets is not well-behaved.

3.2. Compensation

Our algorithm assumes that all subtransactions eventually commit. To deal with the possibility of aborts, we use the notion of compensation [14, 15]. Under this notion, it is assumed that every subtransaction can be compensated. Barring permanent system failures, a compensating subtransaction always commits eventually.

Every subtransaction S keeps track of all nodes to which it sent child subtransactions as well as the node of its parent subtransaction. When S aborts, it rolls back all changes it performed locally and also sends *compensating subtransactions* to the above-mentioned nodes. A compensating subtransaction either causes abort of the corresponding subtransaction (if it has not finished) or rolls back its effects, possibly causing further compensating subtransactions to be sent.[1]

Since we allow subtransactions of the transaction execution tree to visit the same node multiple times, we can view compensating subtransactions as just ordinary members of the global transaction tree. We therefore do not distinguish between compensating and ordinary subtransactions, nor do we differentiate between abort and commit outcomes of subtransactions: a global transaction executes as a tree of subtransactions, which may be ordinary or compensating subtransactions. In particular, for transactions to form a well-behaved set, all their subtransactions (both "normal" and compensating) must commute with each other.

3.3. Inter-Node Version Consistency

We use *global serializability* [4] as our correctness criterion. It says that overall (possibly concurrent) execution of (global) transactions must be equivalent to a serial execution where one transaction is submitted only after the previous one terminates. For a globally serializable execution, a query must not access data that reflects a partially executed update transaction. We accomplish global serializability through the use of versioning at each node, with the following notion of consistency for a version:

Definition 3.2 Inter-node Version Consistency: A version i of data distributed on a set of nodes is consistent across these nodes if the following two conditions hold for every update transaction T that executes against version i on some node: (1) Transaction T has terminated, and (2) *All* subtransactions of T were performed against this version of the data. □

Inter-node version consistency by itself does not entail serializability of transactions completed against this version. It only specifies that the version does not reflect partially executed transactions.

4. The 3V Algorithm

Every node in our algorithm maintains the following variables:

- *vu*, the *current update version number*. When a "root" subtransaction is submitted to a node, it must use records with this version number for updates.
- *vr*, the *current read version number*. Queries must use records with this version number.

A node also maintains up to three active (i.e., used by either queries or update transactions) versions of records. Different versions are distinguished by version numbers. For each active version number v, a node p maintains the following variables:

- $C_{v\,op}$, one *completion counter* for each node o. Counts the number of completed subtransactions submitted to node p from node o against data of version v.
- R_{vpq}, one *request counter* for each node q. Counts the number of subtransactions sent from node p to node q against data of version v.

We assume for simplicity that version numbers increase monotonically with time. A real implementation could re-use old version numbers, employing only three distinct numbers. We also assume that the following questions can be answered efficiently: (1) Does data item x exist in version v? (2) Locate data item x with version v. Finally, our only assumption regarding concurrent accesses to version numbers and counters on a node is that all operations (reads and writes) on these variables are atomic. In particular, when a user transaction in the protocol below reads of writes these variables, these accesses occur outside local concurrency control that serializes all subtransactions on a given node. Thus, these accesses cannot cause any synchronization delays.

Initially, all records exist in a single version 0, the current read version $vr = 0$, and the current update version $vu = 1$. All request and completion counters, C_{0pq} and $C_{0\,op}$, are also 0 for all values of o and q.

4.1. Executing Well-Behaved Update Transactions

When a root subtransaction is submitted to a server, it is assigned a version number, which is then carried by all its descendant subtransactions. The version number associated with subtransaction T will be called the *transaction version number*, and will be denoted by $V(T)$. $\mathsf{source}(T)$ denotes the node that invoked subtransaction T, and $x(v)$ denotes version v of data item x.

Executing a subtransaction T on a node p is done according to the following algorithm (the clean-up phase is not shown):

1. If T is the root subtransaction, assign the current value of vu to be the transaction version number of

[1] However, there is never a need to send more than one compensating subtransaction to any node. Thus, the compensation process will always terminate, at worst after all nodes involved in the global transaction have been visited.

T ($V(T) := vu$). Also, increment the local request counter for node p ($R_{(vu)pp}$) by 1 .

2. If T is not a root subtransaction, and $V(T) > vu$, then advance the update version vu to $V(T)$ ($vu := V(T)$), and allocate and initialize to zero all the request and completion counters for the new version vu ($R_{(vu)op}$ and $C_{(vu)pq}$). Note that exactly the same steps are done in response to getting a start-advancement message during version advancement (Section 4.3).

3. For every data item x that T reads, read the maximum existing version of x that does not exceed version $V(T)$.

4. For every data item x that T updates, check whether $x(V(T))$ exists, and if not, create $x(V(T))$ by copying the value of data item x in the maximum existing version of x that does not exceed version $V(T)$. Checking for the existence and creation (if necessary) of a new version of x is done as one atomic step. Once $x(V(T))$ exists, update *all versions* of x greater or equal to version $V(T)$.

5. Send further subtransactions to other nodes and commit. For every subtransaction sent to node q, increment the request counter $R_{(V(T))pq}$ by 1 before sending the subtransaction.

6. In one atomic step, increment completion counter $C_{(V(T))(\mathsf{source}(T))p}$ by 1, and terminate by committing or aborting. (If T is the root subtransaction, source(T) is equal to p.)

4.2. Executing Read-Only Transactions

The algorithm for executing a read-only transaction (called query for short), is very similar to the algorithm for update transactions presented above.

When a read-only root subtransaction (called query for short) T arrives at node p, it is assigned a version $V(T)$ equal to the value of vr at the time of T's arrival. As in the case of update transactions, this version number is then carried by all descendant query subtransactions. Further, the local request counter $R_{(vr)pp}$ is incremented by 1. (A new transaction has been requested at node p by node p against version vr).

For every data item x that a query subtransaction T reads, the node reads the maximum existing version of x that does not exceed $V(T)$. (Thus, as in the case of update transactions, the same subtransaction may read data items of different versions, depending on which versions of each data item exist.)

For every query subtransaction sent to node q, increment the request counter $R_{(V(T))pq}$ by 1 before sending the subtransaction.

When T completes, in one atomic step, increment completion counter $C_{(V(T))(\mathsf{source}(T))p}$ by 1, and commit. (If T is the root query subtransaction, source(T) is equal to p.)

4.3. Version Advancement and Garbage Collection

We assume for simplicity that a distributed mutual exclusion mechanism is employed to ensure that at most one instance of the version advancement process can run at any time. In particular, when the algorithm is initiated, all nodes have the same read version number, the same update version number, and $vr = vu - 1$ on all nodes.

Let vr_{old} and vu_{old} be read and update version numbers kept on nodes before the version advancement begins. Advancement to a new read version is done in four phases:

Phase 1. Switching to a new update version: The coordinator sends a message "start-advancement" with the new update version number $vu_{new} = vu_{old} + 1$ to every node. Each recipient of this message advances its update version to vu_{new} (Sets $vu = vu_{new}$), allocates and initializes to zero all the request and completion counters for the new version ($R_{(vu)op}$ and $C_{(vu)pq}$), and responds with an acknowledgement.

Once every node has acknowledged advancing its update version, it is guaranteed that all new transactions will have a version number vu_{new}, and will update records of version vu_{new}. Then the coordinator initiates Phase 2.

Phase 2. Updates phase-out: The coordinator waits until data of version vu_{old} become mutually consistent across all nodes (Definition 3.2). To determine consistency, the coordinator reads, in an asynchronous manner, the request and execution counters at each node. It can be shown that if $R_{(vu_{old})pq} = C_{(vu_{old})pq}$ for all nodes p and q, then version vu_{old} is consistent across all nodes. Once this condition is satisfied, the coordinator initiates Phase 3.

If the counters do not match up, they are read again. Since all new transactions are assigned version number vu_{new} and all transactions eventually terminate, it is guaranteed that data with any earlier version number will eventually become mutually consistent.

Phase 3. Switching to a new read version: Since data of version vu_{old} is now mutually consistent across all nodes, and all updates are directed to version $vu_{new} = vu_{old} + 1$, queries can now be allowed to execute against data of version $vu_{old} = vr_{old} + 1$. Therefore, the coordinator sends a new read version number $vr_{new} = vr_{old} + 1$ to all nodes. A node that receives this message advances its read version vr to vr_{new}, and starts executing new root queries using version vr_{new}. Once this phase completes, all newly arrived queries have version number vr_{new}.

Phase 4. Garbage Collection:. The coordinator waits until all queries with the old version vr_{old} terminate. Termination is checked by an asynchronous read of the request and execution counters at each node. If $R_{(vr_{old})pq} = C_{(vr_{old})pq}$ for all nodes p and q, then the co-ordinator determines that all queries against version vr_{old} have terminated.

Once the counters match up, the coordinator notifies every node that it should garbage-collect old versions. Upon receiving this notification, a node garbage collects as follows: For every data item x, if version $x(vr_{new})$ exists, the node garbage-collects all earlier versions of this record. If version $x(vr_{new})$ does not exist, the node changes the version number of the latest earlier version of x to vr_{new}. Finally, it garbage-collects all counters associated with version numbers smaller than vr_{new}.

Note that comparing request and completion counters detects transaction termination correctly even if some subtransactions abort and invoke compensation. Indeed, compensating subtransactions are not distinguished from other subtransactions in our model. Therefore, according to the protocol of Section 4.1, a node sending a compensating subtransaction to another node first increments the corresponding request counter. So, even in a situation when all "normal" subtransactions have terminated and messages with compensating subtransactions are in transit, the request and execution counters will not be equal for the nodes to which compensating subtransactions have been sent, meaning that the whole transaction has not completed yet.

4.4. Properties and Correctness of the 3V Algorithm

The 3V algorithm described above satisfies the following properties.

1. While the version advancement process is not running:
   (a) Only two versions of each data item exist.
   (b) The read version number vr is the same at all nodes.
   (c) The update version number vu is the same at all nodes.
2. While the version advancement process is running:
   (a) At most three versions of each data item exist.
   (b) if two sites p and q differ on their update version numbers, then they must have the same read version numbers. Conversely, if two sites p and q differ on their read version numbers, then they must have the same update version numbers.
3. The update version number is always greater than the read version number. Further, $vr < vu \leq vr + 2$.
4. A transaction with an update version i may update data of versions i and $i + 1$. It may read only the data of version i or less.
5. Once all the nodes advance their update version number from vu_{old} to vu_{new}, the property "Is it true that all update transactions on version vu_{old} have terminated" is a stable property [12], meaning that once it becomes true, it stays true in all subsequent states.

The 3V global concurrency control algorithm is used in conjunction with local concurrency control algorithms on each node that guaranty serializability of the subtransactions submitted locally at that node. Under the above conditions, we can show that the 3V protocol is correct, and does not require global synchronization between subtransactions.

Theorem 4.1 *Every schedule produced by the 3V algorithm is equivalent to any 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.* □

Theorem 4.2 *Let subtransactions be executed under the 3V algorithm, and let version advancement be done according to 3V algorithm. Then, no subtransaction ever waits for any non-local subtransaction, or for any activity (local or non-local) related to version advancement.* □

5. Non-Commuting Updates

Thus far we have required that all update subtransactions be well-behaved (commute). In this section, in addition to the well-behaved set of update transactions, and the set of read-only transactions, we allow additional update transactions that do not commute with each other or with subtransactions of well-behaved update transactions.

The extension to the basic algorithm is as follows. We require that the well-behaved update transactions acquire special *commuting-update* and *commuting-read* locks on all data they access, and retain these locks according to the classical two-phase locking algorithm for distributed concurrency control. The well-behaved transactions have no global commitment, and a special clean-up phase is required to go through and release all commute locks after all subtransactions in a transaction tree have committed. The clean-up is asynchronous with respect to well-behaved transactions. The clean-up is asynchronous with respect to well-behaved transactions. Non-well-behaved transactions are required to obtain non-commuting locks on records they access and also follow two-phase locking. These non-commuting update transactions also need to perform two-phase global commitment.

Commuting locks are compatible with each other but not with their non-commuting counterparts. Thus, in the absence of non-well-behaved transactions, there is no wait to obtain a commute lock, and the performance of the system does not suffer. However, non-well-behaved transactions are serialized in the same way as traditional transactions are serialized in distributed databases.

A root non-well-behaved subtransaction waits until the local value of vu is equal to $vr + 1$ before executing. The only time these values do not differ by one is during the version advancement process, since vu is incremented first thing in phase 1 and vr only in phase 3. By these means we make sure that non-well-behaved transactions do not conflict with version advancement.

Putting all of this together, a node p processes a non-well-behaved root update subtransaction K according to the following NC3V algorithm (for simplicity, locking of data is not shown):

1. Set K's version number to the current value of vu: $V(K) = vu$.

2. If $V(K) = vr + 1$, proceed. Else, wait until the condition $V(K) = vr + 1$ becomes true. This step ensures that a non-well-behaved transaction may proceed in a given version only after all update transactions (both well-behaved and not) that executed in earlier versions have finished.

3. For every data item x that K reads, read the maximum existing version of x that does not exceed $V(K)$.

4. For every data item x that K updates, check whether x exists in a version greater than $V(K)$, in which case abort K. Otherwise, check whether $x(V(K))$ exists, and if not, create it. Checking for the existence and creation (if necessary) of $x(V(K))$ is done as one atomic step. Then, update $x(V(K))$.

5. Send further subtransactions to other nodes. For every subtransaction sent to node q, increment the local request counter, $R_{(V(K))pq}$ by 1.

6. Participate in a global two-phase commitment. (The completion counter, $C_{(V(K))(\mathsf{source}(K))p}$, is incremented by 1 atomically together with commitment.)

The algorithm for processing a non-well-behaved non-root subtransaction is the same except steps 1 and 2 are not needed. The algorithm for processing well-behaved and read-only transactions is the same as the 3V algorithm. To see this, observe that, due to step 2 of the NC3V algorithm, a well-behaved transaction executing in version i can never encounter a data item of a greater version that is written by a non-well-behaved transaction.

Theorem 5.1 *The NC3V algorithm produces a global schedule of subtransactions that is serializable provided each node guarantees local serializability.* □

6. Application Domain – Data Recording Systems

The assumptions we have made in developing the 3V algorithm are (1) update subtransactions commute within themselves, (2) reads and updates do not commute with each other, (3) application is spread over multiple nodes, (4) global synchronization between nodes is not desirable, and (5) global consistency is desirable.

We now describe a class of large distributed systems called data recording systems [10, 13] where the above assumptions hold. Examples of data recording systems include (a) operation monitoring systems in automated factories, oil pipelines, and computer networks, (b) information gathering systems, such as in the case of satellites, and (c) transaction recording systems for credit card transactions, telephone calls, stock trades, and flight reservations.

A fundamental characteristic of the above data recording systems is that they record data by inserting new data observations into a database, and simultaneously update summaries (such as account balances, number of items sold, number of parts produced) that are derived from the recorded data [13]. The recorded data itself is modified infrequently, if at all (except for a clean-up or archival process that periodically removes old data from the database). However the summaries are updated upon each recording activity. Where updates simply record new information by inserting tuples and updating summaries, the final state of the database is the same after the application of two updates, irrespective of the order which they were applied. Also, the value returned by the update transaction, if any, is independent of the ordering of updates. Thus, most update subtransactions in data recording systems commute locally. At the same time, read operations can read summaries, and do not commute with the updates.

Data recording systems usually deal with very large volumes of data. For example, AT&T's call recording system records several million calls every hour. Due to their size, or even due to administrative reasons in a company, these systems need to be distributed across multiple databases on multiple machines, sometimes in geographically diverse areas. Due to the high performance needs, it is impractical to run global synchronization of transactions across multiple nodes. At the same time, global consistency is desirable for customer enquiries and auditing queries.

7. Related Work

There has been much work to avoid interference between read-only and update transactions by creating multiple versions of data on updates [18, 7, 19, 1, 17]. In fact, such schemes have even been implemented in products, going back to Prime Computer's CODASYL system and Digital's

Rdb/VMS. However, each of these schemes either requires global coordination between some transactions, or between transactions and the version advancement mechanism. By exploiting the commutativity of (most) update transactions, we are able to avoid these coordination overheads.

Techniques suggested in [6, 7, 1, 5] require each update transaction to create a new version. This entails copying an entire data object on every update, no matter how small the modification. On the other hand, queries always use the latest version that can be read without violating serializability. Our protocol provides flexibility in trading data currency of queries for update performance by giving the user control over when version advancement should occur. Since data copying in our protocol occurs only once after version advancement, the user can improve update performance by increasing the time between version advancements (and allowing queries to read potentially older data).

[1] 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. In this respect, our algorithm borrows from that work, since our versioning scheme is also independent of any specific concurrency control method.

The methods presented in [19] require that read-only transactions update meta-information of data items they read. This may add significant performance overhead to the system, and results in a need for synchronization w.r.t. meta-information.

In [17], as in the 3V algorithm, a user controls how far behind queries get, by determining when version advancement is initiated. However, unlike the 3V algorithm, version advancement in [17] is not coordinated among nodes, and transactions are required to perform a global commit and verify that there was no version mismatch. Also, four versions of data are required, whereas we need at most three.

The algorithms of [3, 18] never keep more than two versions of data, while our algorithm may keep up to three versions. However, in [3], a read-only query may delay the commitment of an update transaction. An update transaction may be aborted because of the versioning mechanism. Also, read-only transactions, while never delayed, incur overhead of obtaining read locks and executing a complex algorithm to decide which data version to use. In addition, this scheme needs to maintain a centralized graph of transaction dependencies, which makes its the applicability to distributed systems unclear. The algorithm of [18] may also delay update transactions. Moreover, even queries can be delayed or aborted in some cases.

Semantics-based concurrency control approaches have been proposed, that take into account commutativity of individual operations [2]. The problem we discuss involves read and update operations that do not commute. Hence, the algorithms of [2] cannot be used to solve our problem.

Committing a subtransaction early and then performing compensation if required is reminiscent of a saga [11]. The key difference between sagas and our work is that, in the case of sagas, intermediate results from committed subtransactions are immediately exposed to all other transactions, including read transactions. Thus, execution of sagas is in general not serializable. While such relaxation of correctness may be necessary for systems with long-running transactions (like workflows), our targeted application domain requires serializability.

In summary, while there has been much work on reducing global coordination in distributed databases, the previous work focuses on general transactions, or assumes all transactions commute. We observed that for a large class of applications in data recording systems, update transactions commute amongst themselves, although not with read-only transactions. We have exploited the commuting property and the asynchronous version advancement mechanism proposed here to provide global serializability without any global synchronization.

8. Conclusions

Data recording systems are ubiquitous, and often have high performance requirements. Where recording systems are distributed, coordinating the updates at multiple nodes is a challenge. We observe that most or all update subtransactions in such systems commute with each another, but do not commute with read subtransactions. A multi-version concurrency control alagorithm can be used to exploit this commutativity and guarantee global serializability without requiring any global synchronization. However, version advancement remains an issue. We presented such an algorithm, where even version advancement and garbage collection can be carried out without incurring any synchornization overhead for user transactions. Specifically, the 3V algorithm is characterized by the following properties:

- Read-only transactions are never delayed or aborted. They do not need to obtain any locks or record any control information.

- In the periods when non-commuting update subtransactions do not execute, no user transaction (read or update) on a node can be delayed by any activity (system- or user-initiated) on other nodes.

- The number of versions of any data item is limited to three.

- Version advancement is performed asynchronously with user transactions.

The 3V algorithm can be useful in federated databases applications where update transactions on member databases commute. In a federated database, each individual node may be running its own transaction manager, so that accomplishing a global transaction with a coordinated commitment or global concurrency control becomes impossible without violating autonomy of the local transaction managers. Yet, we would like to obtain global serializability in the execution schedule. The 3V algorithm can provide the global serializability property.

Acknowledgments

The authors thank Avi Silberschatz for discussions and helpful comments, and Mike Merritt for pointers to the relevant literature on distributed termination detection.

References

[1] D. Agrawal and S. Sengupta. Modular Synchronization in Multiversion Databases: Version Control and Concurrency Control. In *Proceedings of the 1989 ACM SIGMOD International Conference on Mangement of Data*, pages 408–417. Association for Computing Machinery, June 1989.

[2] B. R. Badrinath and K. Ramamritham. Semantics-based concurrency control: Beyond commutativity. *ACM Transactions on Database Systems*, 17(1):163–199, Mar. 1992.

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

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

[5] P. M. Bober and M. J. Carey. On mixing queries and transactions via multiversion locking. In *Proceedings of the Seventh IEEE International Conference on Data Engineering*, pages 535–545, Phoenix, AZ, February 1992.

[6] 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 M. Schkolnick, editor, *Proceedings of ACM SIGMOD 1982 International Conference on Management of Data*, pages 184–191, Orlando, FL, June 2-4 1982.

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

[8] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed system s. *ACM Transactions on Computer Systems*, 3(1):63–75, January 1985.

[9] K. M. Chandy and J. Misra. An example of stepwise refinement of distributed programs: Quiescence detection. *ACM TOPLAS*, 8(3), July 1986.

[10] A. Datta. Research Issues in Databases for ARCS: Active Rapidly Changing Data Systems. *Sigmod Record*, 23(3):8–13, 1994.

[11] H. Garcia-Molina and K. Salem. Sagas. In U. Dayal and I. Traiger, editors, *Proceedings of ACM SIGMOD 1987 International Conference on Management of Data*, pages 249–259, San Francisco, CA, May 27-29 1987.

[12] 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 (PODC)*, pages 125–136, August 1987.

[13] H. V. Jagadish, I. S. Mumick, and A. Silberschatz. View maintenance issues in the chronicle data model. In *Proceedings of the Fourteenth Symposium on Principles of Database Systems (PODS)*, pages 113–124, San Jose, CA, May 22-24 1995.

[14] H. F. Korth, E. Levy, and A. Silberschatz. A formal approach to recovery by compensating transactions. In D. McLeod, R. Sacks-Davis, and H. Schek, editors, *Proceedings of the Sixteenth International Conference on Very Large Databases*, pages 95–106, Brisbane, Australia, August 13-16 1990.

[15] E. Levy, H. F. Korth, and A. Silberschatz. An optimistic commit protocol for distributed transaction management. In *Proceedings of ACM SIGMOD 1991 International Conference on Management of Data*, pages 88–97, Denver, CO, May 29-31 1991.

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

[17] C. Mohan, H. Pirahesh, and R. Lorie. Efficient and flexible methods for transient versioning of records to avoid locking by read-only transactions. In *Proceedings of ACM SIGMOD 1992 International Conference on Management of Data*, pages 124–133, San Diego, CA, June 2-5 1992.

[18] R. E. Stearns and D. J. Rosenkrantz. Distributed database concurrency controls using before-values. In *Proceedings of the 1981 ACM SIGMOD International Conference on Mangement of Data*, pages 74–83. Association for Computing Machinery, 1981.

[19] W. E. Weihl. Distributed version management for read-only actions. *IEEE Transactions on Software Engineering (TSE)*, 13(1):55–64, Jan. 1987.