Update Propagation Protocols For Replicated Databases

Yuri Breitbart[1] **Raghavan Komondoor**[2*] **Rajeev Rastogi**[1] **S. Seshadri**[1]
Avi Silberschatz[1]
[1] Bell Laboratories, Murray Hill, NJ 07974
[2] University Of Wisconsin, Madison, WI 53706
{yuri,rastogi,seshadri,avi}@research.bell-labs.com
raghavan@cs.wisc.edu

Abstract

Replication is often used in many distributed systems to provide a higher level of performance, reliability and availability. Lazy replica update protocols, which propagate updates to replicas through independent transactions after the original transaction commits, have become popular with database vendors due to their superior performance characteristics. However, if lazy protocols are used indiscriminately, they can result in non-serializable executions. In this paper, we propose two new lazy update protocols that guarantee serializability but impose a much weaker requirement on data placement than earlier protocols. Further, many naturally occurring distributed systems, like distributed data warehouses, satisfy this requirement. We also extend our lazy update protocols to eliminate all requirements on data placement. The extension is a hybrid protocol that propagates as many updates as possible in a lazy fashion. We implemented our protocols on the Datablitz database system product developed at Bell Labs. We also conducted an extensive performance study which shows that our protocols outperform existing protocols over a wide range of workloads.

1 Introduction

Distributed applications frequently use replication as a means to achieve a higher level of performance, reliability and availability. Consequently, the management of replicated data has emerged as a problem of great practical importance in recent years. Gigabytes of data are replicated in distributed data warehouses and various web sites on the internet. In telecom as well as data networks, network management applications require real-time dissemination of updates to replicas with strong consistency guarantees.

*This work was done while the author was visiting Bell Labs.

There are two broad approaches to handle the problem of replica updates in a distributed database system viz., *eager* and *lazy*. The eager approaches update all the replicas of an item as part of a single transaction. Thus, eager approaches ensure that executions are serializable. However, a major disadvantage of eager algorithms is that the number of operations in the transaction increases with the degree of replication, and since deadlock probability is proportional to the fourth power of the transaction size, eager protocols are unlikely to scale beyond a small number of sites [GHKO81, GHOS96]. In contrast, lazy update propagation algorithms post updates to replicas through independent transactions that are spawned by the original updating transaction after it commits. Thus, the effective size of a transaction is reduced and the overall performance of the system improves due to fewer deadlocks; however, transaction execution has to be orchestrated carefully to ensure global serializability.

Due to its superior performance benefits, a number of database vendors (e.g, Sybase, Oracle, CA-OpenIngres) provide support for lazy replication. Specifically, they provide an option in which each transaction executes locally, and then asynchronously propagates its updates to replicas after it commits (the replicas at each site are updated in the context of a separate transaction). Since each transaction executes locally and independently, the systems do not require multi-site commit protocols (e.g., two-phase commit) which tend to introduce blocking and are thus not easily scalable. A problem, however, with the lazy replication approaches of most commercial systems is that they can easily lead to non-serializable executions. For instance, it is possible for the same data item to be concurrently updated at two different sites, thus resulting in an update conflict. Currently, commercial systems use reconciliation rules (e.g., install the update with the later timestamp) to merge conflicting updates. These rules do not guarantee serializability unless the updates

are commutative.

In this paper, we propose two new lazy update protocols that guarantee serializability but impose a much weaker requirement on data placement than earlier protocols for ensuring serializability. We also extend our lazy update protocols to eliminate all requirements on data placement. The extension is a hybrid protocol that propagates as many updates as possible in a lazy fashion. Before discussing our specific contributions in more detail, we present the system model adopted in this paper and prior work from [CRR96, GHOS96, BK97, ABKW98].

1.1 System Model

The model we adopt in this paper is very similar to one from [CRR96, BK97]. For each data item, a particular site is chosen as its *primary site*. The copy of a data item at the primary site is called the *primary copy* and the other copies are referred to as *secondary copies* or *replicas*. Each transaction originates at a single site and is a sequence of read and write operations. A transaction can read all data items at its originating site; however, it is permitted to update only data items whose primary copies are at the transaction's originating site. We assume that each site follows the strict two-phase locking (2PL) locking protocol to ensure that local executions are serializable[1]. We also assume that the underlying network delivers messages reliably and in FIFO order between any two sites.

We define a *copy graph* to be the directed graph in which the set of vertices corresponds to the set of sites. An edge from site s_i to site s_j exists in the copy graph if and only if there exists an item whose primary copy is at site s_i and one of whose secondary copies is at site s_j. A set of edges in the copy graph are referred to as *backedges* if their deletion breaks all cycles in the copy graph.

Also, we will say that a transaction propagates updates *lazily* if 1) it does not communicate with any remote site (e.g., for the purpose of obtaining locks) during its execution and until it commits, and 2) it's updates are propagated asynchronously to sites as independent transactions only after it has committed and released all its locks[2].

1.2 Lazy Update Protocols – Existing Approaches

The problem of ensuring serializability in the lazy update model was first addressed in [CRR96]. The authors obtain a tight characterization of global serializability based on the topology of data distribution when replica updates are propagated lazily and *indiscriminately*. Specifically, they show that lazy replication protocols guarantee serializability if and only if the undirected graph obtained from the copy graph (by removing the directions on the edges) is acyclic. However, since replica updates are propagated arbitrarily, their algorithms cannot ensure serializability if the copy graph is a *directed acyclic graph* (DAG), as illustrated by the following example.

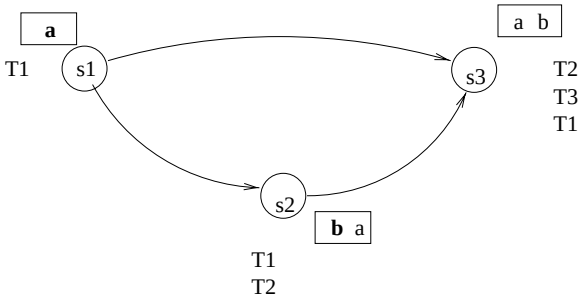


Figure 1: Example of Non-serializable Execution

Example 1.1 Consider the distributed system, shown in Figure 1, with 3 sites and two items a and b. The primary site for a is s_1 with secondary copies at s_2 and s_3, while the primary site for b is s_2 and a secondary copy is at s_3, and the copy graph is as shown in Figure 1. Consider 3 transactions, T_1 at site s_1, T_2 at site s_2 and T_3 at site s_3. T_1 updates item a, T_2 reads a and writes b while T_3 reads both items a and b. Assuming lazy propagation of updates to replicas, it is possible for T_1's update to reach site s_2 before T_2 executes, but to reach site s_3 after T_2's update to b has been applied and transaction T_3 has completed execution. The resulting execution is non-serializable since T_1 is serialized before T_2 at site s_2, but T_2 is serialized before T_1 at site s_3. □

In addition to the work from [CRR96], the problem of replica update propagation is also addressed in [GHOS96] and [BK97, ABKW98]. However, neither approach is lazy according to our definition. In [GHOS96], any transaction that wishes to read or write a primary copy or replica of an item is required to get an appropriate lock from the item's primary site. In addition, in order to guarantee serializability, locks on items that are updated need to be held until all the updates have been propagated to all the replicas.

In [BK97, ABKW98], the authors describe a protocol, which maintains a *replication* graph that contains information about the execution of every transaction in the distributed system. They describe how to maintain this graph at a centralized site but also observe that the

[1] The variant of 2PL we assume is one in which a transaction does not release any locks (read or write) until after it has committed.

[2] Note that our definition of lazy protocols differs slightly from the definitions adopted in [GHOS96, BK97, ABKW98].

central site becomes a bottleneck if the number of sites becomes large.

1.3 Our Contributions

In this paper, we make two major contributions. The first contribution of this paper is that we develop two new lazy update protocols, DAG(WT) and DAG(T), which guarantee serializability as long as the copy graph is a DAG. The second contribution of this paper is the BackEdge protocol, which augments the DAG(WT) and DAG(T) protocols above, by eliminating the requirement that the copy graph be a DAG. We briefly describe the contributions below.

The DAG(WT) and DAG(T) protocols: The DAG(WT) and DAG(T) protocols guarantee serializability as long as the copy graph is a DAG. Therefore, compared to [CRR96], we significantly expand the class of copy graphs for which lazy update protocols can be made to produce serializable schedules. Both protocols ensure serializability by controlling the order in which updates to secondary copies are applied at sites.

The BackEdge protocol: The BackEdge protocol is a *hybrid* protocol that combines the eager and lazy approaches, performing eager update propagation along backedges while following one of the DAG lazy update protocols for update propagation along other edges (which form a DAG). Thus, locks for items updated along backedges are acquired at multiple sites and held until the transaction commits, while updates along the remaining edges are relayed asynchronously and lazily after the transaction has committed.

Using our DAG and BackEdge protocols, database designers can thus guarantee serializable executions in replicated environments by adding some minimal code (corresponding to our protocols) on top of off-the-shelf database systems. We implemented a simpler variant of the BackEdge protocol (extension of DAG(WT)) and also a lazy version of the primary site locking algorithm (which is a variant of the lazy-master approach from [GHOS96]) on the DataBlitz database system product developed at Bell labs [BLRSSS97]. The results of our extensive performance study indicate that the BackEdge protocol consistently outperforms the primary site locking algorithm except for the extreme case involving update-intensive transactions and a copy graph with a large number of backedges. For most practical environments, in which transactions are read-intensive and backedges are few, the BackEdge protocol achieves speedups as high as five times compared to the primary site locking protocol.

2 DAG(WT) Protocol

In this section, we describe the DAG(WT) (DAG Without Timestamps) protocol for acyclic copy graphs. As shown in Example 1.1, disseminating replica updates indiscriminately could result in non-serializable executions even though the copy graph is a DAG. In order to rectify this problem, the DAG(WT) protocol propagates replica updates along the edges of a tree T constructed from the DAG corresponding to the copy graph. The tree T constructed has the property that if a site s_i is a child of site s_j in the copy graph, then s_i is a descendant of s_j in the tree T. In [BKRSS98], we outline how to construct a tree T with the above property and omit those details here for the sake of brevity.

In the DAG(WT) protocol, a transaction executes at a single site s_i and the transaction's updates are forwarded to the children of s_i in the tree T. Thus, at any site there are transactions that originated at the site, referred to as *primary* subtransactions, and there are transactions (consisting of a primary subtransaction's updates) that were forwarded to it by it's parent, called *secondary* subtransactions. Updates for items in a secondary subtransaction received at a site are applied only for those items with replicas at the site – as a result, it is possible that a secondary subtransaction may perform no updates at a site. Furthermore, the forwarded secondary subtransactions from a parent are committed at a site in the order in which they are received at the site, and are in turn forwarded to the site's children. Finally, the forwarding of both primary as well as secondary subtransactions at a site is done atomically with respect to commit, that is, if T_i commits before T_j at a site, then T_i is forwarded before T_j to it's children.

Actually, when a subtransaction commits at site s_i, secondary subtransactions need to be forwarded only to the *relevant* children of s_i rather than all children. A child is relevant for a subtransaction if either the child or one of it's descendants contains a replica of an item that the subtransaction has updated.

Thus, it follows that the DAG(WT) protocol is a lazy update protocol since each transaction executes completely locally until it commits and releases all its locks at commit time. The DAG(WT) protocol is similar to the *tree* protocol [SK80] – however, unlike the tree protocol, it operates at the granularity of sites and not items. By propagating secondary subtransactions sequentially along the edges of the tree T, it ensures that when a secondary subtransaction for a transaction is executed at a site, all transactions preceding it in the serialization order have committed at the site. A proof of the following theorem can be found in [BKRSS98].

Theorem 2.1 *Any schedule produced by the DAG(WT) protocol is serializable.*

At each site, the secondary subtransactions are committed in the order they are received and therefore as long as subtransactions received earlier than a particular subtransaction T_i commit, eventually, T_i will also commit. If the subtransaction that is received earlier than T_i gets aborted after it starts execution at a site due to a local deadlock at that site, the subtransaction will have to be repeatedly resubmitted until it succeeds. To guarantee that this subtransaction is not chosen as a victim of a deadlock all the time, some fair victim selection policy, e.g., the transaction which arrived at the site the latest, will have to be used. Therefore, eventually T_i will complete.

The non-serializable execution in Example 1.1 will not be permitted by the DAG(WT) protocol. This is because in the tree T^3 that satisifies the desired property, site s_3 is a child of site s_2 which in turn is a child of site s_1. As a result, the update of a by transaction T_1 cannot be directly sent to site s_3, but would have to be first sent to site s_2 and then site s_2 would forward it to site s_3. Since T_1's update to a executes before T_2 at site s_2, T_1's update would be forwarded to site s_3 before T_2's update to b. Thus, T_1 would be serialized before T_2 at site s_3.

3 The DAG(T) protocol

The DAG(WT) protocol propagated updates only along the edges of the tree T. The problem with this is that a secondary subtransaction may need to be routed through a number of other intermediate sites at which it has no updates to perform, before it can be executed at its destination site. As a result, the DAG(WT) protocol could result in significant messaging overhead in the network and processing costs at sites. Furthermore, transaction updates could experience unnecessary propagation delays.

In this section, we describe the DAG(T) (DAG with Timestamps) protocol that propagates updates along the edges of the copy graph itself. As a result, updates can now be directly sent to the relevant sites rather than routing them through intermediate nodes as was done in Section 2. However, the perils of propagating updates along edges of the copy graph, without any additional controls, are evident from Example 1.1.

Instead of superimposing a structure on the copy graph to control update propagation, as was done by the DAG(WT) protocol, the DAG(T) protocol employs *timestamps* to impose an order on secondary subtransaction execution. In the DAG(T) protocol, primary subtransactions are assigned a system wide unique timestamp when they commit, and the secondary subtransactions carry this timestamp with them to the secondary sites. At each site, transactions are executed in timestamp order to ensure serializability. The crux of the protocol is in suitably defining this timestamp so that timestamps themselves are totally ordered, and then ensuring that the protocol executes transactions in timestamp order. Moreover, the timestamp also needs to be augmented to ensure that if a primary subtransaction has completed, then all it's secondary subtransactions will eventually execute at the secondary sites. The preceding progress criteria does not follow automatically from the fact that transactions are executed in timestamp order. In fact, a protocol that trivially guarantees serializability but does not guarantee progress is one in which no secondary subtransaction is allowed to execute anywhere in the system! We address this issue in Section 3.3.

3.1 Timestamps

Since the copy graph is acyclic, there exists a total order $<$ on the sites. Without loss of generality, let the total order on the sites be $s_1 < s_2 < \cdots < s_m$. At each site s_i, a local timestamp counter denoted by LTS_i (initially 0), is maintained which keeps track of the number of primary subtransactions that have committed at s_i. We will first define the notion of a tuple which forms the building block for constructing the timestamp of a transaction.

Definition 3.1 A tuple corresponding to site s_i is an ordered pair (s_i, LTS_i). □

One option is to simply use the tuple for site s_i as the timestamp for a transaction when it commits at site s_i. However, only the tuple for a site does not capture information about the serialization order of transactions and is thus inadequate as a timestamp. To illustrate, consider Example 1.1. Suppose we were to assign timestamps using the tuple for a site, then T_1 would be assigned the timestamp $(s_1, 1)$ (the tuple for site s_1 when it commits), and T_2, the timestamp $(s_2, 1)$, irrespective of the order in which T_1 and T_2 commit at site s_2. Thus, simply from the timestamps, it is impossible to determine the order in which T_1 and T_2 must be executed at site s_3, which leads us to the following more elaborate definition for timestamps.

Definition 3.2 The timestamp of a site s_i, denoted by $TS(s_i)$, is a vector of tuples – the vector contains a tuple for the site s_i itself, and every other tuple in the vector belongs to an ancestor of s_i in the copy graph. The tuples within the vector are ordered based on the sites. In other words, the tuple for s_j appears before s_l in the vector if and only if $s_j < s_l$. □

[3] In general, for arbitrary DAGs, there may be several trees satisfying the desired property

Note that the timestamp for a site contains tuples for some subset of its ancestors, and not necessarily for every ancestor. The timestamp of a transaction T_i, denoted by $TS(T_i)$, is the timestamp at the site where the primary subtransaction executed when it committed. Since, a tuple is a vector consisting of a site and an integer counter, and sites are ordered, tuples can be compared lexicographically. Based on this lexicographic ordering of tuples, we can lexicographically order timestamps (which are nothing but a vector of tuples). Formally, we define a lexicographic ordering $<$ on timestamps as follows:

Definition 3.3 Let TS_1 and TS_2 be two distinct timestamps. We define $TS_1 < TS_2$ if and only if

- TS_1 is a prefix of TS_2, or
- Let $TS_1 = X(s_i, LTS_i)Y_1$ and $TS_2 = X(s_j, LTS_j)Y_2$, that is, TS_1 and TS_2 share a common prefix X of tuples and the first pair of tuples that they differ on is (s_i, LTS_i) and (s_j, LTS_j). Then, $TS_1 < TS_2$ if one of the following is true:
  1. $s_i > s_j$, or
  2. $s_i = s_j$ and $LTS_i < LTS_j$.

□

Note that in the lexicographic ordering of tuples used in the above definition, we use the reverse ordering for the sites. Thus, according to Definition 3.3

1. $(s_1, 1) < (s_1, 1)(s_2, 1)$
2. $(s_1, 1)(s_3, 1) < (s_1, 1)(s_2, 1)$
3. $(s_1, 1)(s_2, 1) < (s_1, 1)(s_2, 2)$.

It is straightforward to observe that Definition 3.3 defines a total order $<$ on timestamps. Note that tuples within a timestamp still appear in the order of their sites – only when comparing two timestamps do we use the reverse ordering on sites. The motivation for reordering sites for the comparison of two timestamps will become clearer as we describe the protocol – however, the intuition, at a very high level, is the following. Again, we go back to Example 1.1. Intuitively, the timestamp of a transaction is used to capture information about transactions serialized before it. Thus, when T_2 commits at s_2 after T_1's update to a has been applied at s_2, it is assigned a timestamp $(s_1, 1)(s_2, 1)$ where the first tuple $(s_1, 1)$ is used to capture the fact that T_1 with timestamp $(s_1, 1)$ is serialized before it. Consequently, at site s_3, since T_1's timestamp is smaller than T_2's it will be committed earlier. However, it is possible that some other transaction T_3 may commit right after T_1 committed and before T_2 committed – this transaction would be assigned a timestamp $(s_1, 1)(s_3, 1)$ and is serialized before T_2 at site s_3. Since primary subtransactions can commit immediately (due to our lazy update propagation assumption) and we would like transactions to commit in the order of their timestamps at each site, we are forced to define the ordering on timestamps as in Definition 3.3 according to which $(s_1, 1)(s_3, 1) < (s_1, 1)(s_2, 1)$.

3.2 The DAG(T) Protocol

Having defined the structure of timestamps, we will now describe the DAG(T) protocol in three parts, the data structures that need to be maintained, the actions of a primary subtransaction and the actions of a secondary subtransaction.

3.2.1 Data Structures

The first data structure maintained at a site is the timestamp vector of the site – this is simply the concatenation of the timestamp of the last secondary subtransaction that committed at the site and the tuple for the site. Initially, $TS(s_i)$ is set to $(s_i, 0)$ for every site s_i. The second data structure is a set of incoming queues. One incoming queue is maintained per parent of the site in the copy graph. The queue contains secondary subtransactions that are waiting to be executed at the site.

3.2.2 Primary Subtransactions

Primary subtransactions can start executing as soon as they are submitted. When a primary subtransaction T_i commits at site s_i the following steps are taken:

1. Increment the local timestamp LTS_i of s_i in the tuple corresponding to s_i in $TS(s_i)$, the timestamp vector of the site s_i.
2. Set $TS(T_i) = TS(s_i)$. This is the timestamp for the primary subtransaction and all its secondary subtransactions.
3. Schedule secondary subtransactions of T_i at all the relevant children of s_i. The scheduling here basically means that a message containing the list of writes that the primary subtransaction has performed along with $TS(T_i)$ is appended to the incoming queues of s_i's children. A child is relevant for a subtransaction if that child contains a replica of an item that this subtransaction has updated.

The above three steps are performed in a critical section to ensure that no other subtransaction concurrently commits and manipulates the local timestamps and the message queues.

3.2.3 Secondary Subtransactions

We assume for simplicity that only one secondary subtransaction is executed at a time at any given site (this assumption can be easily relaxed). Each secondary can execute concurrently with any number of primary subtransactions at a site. Further, at any point, the secondary subtransaction with the minimum timestamp from among the subtransactions at the head of each incoming queue at a site is chosen for execution next. Note that there must be at least one subtransaction in each incoming queue before the minimum timestamp is computed. When a secondary subtransaction T_i commits at a site s_i, $TS(s_i)$ is set to $TS(T_i)(s_i, LTS_i)$, which is simply the concatenation of the timestamp of the subtransaction and the tuple for the site (s_i, LTS_i). The above commit and setting of the timestamp is done atomically with respect to commits of other subtransactions.

Thus, the DAG(T) protocol propagates updates lazily – each transaction executes locally and independently, and releases all its locks when it commits. The DAG(T) protocol would not allow the non-serializable execution from Example 1.1. In this case, T_1 is assigned a timestamp of $(s_1, 1)$ and T_2 gets assigned a timestamp of $(s_1, 1)(s_2, 1)$ (when T_1 commits at s_2, the site's timestamp is set to $(s_1, 1)(s_2, 0)$). As a result, at site s_3, since T_1's timestamp is a prefix of T_2's, T_1 will be executed before T_2 at site s_3 preventing the non-serializable execution from Example 1.1.

Intuitively, the timestamp for a transaction captures information about other transactions serialized before it. We show that, at each site, transactions commit in the order of their timestamps in [BKRSS98]. Since the transaction timestamps are totally ordered and subtransactions are serialized in commit order due to the strict 2PL assumption, the following theorem follows.

Theorem 3.1 *Any schedule produced by the DAG(T) protocol is serializable.*

3.3 Extensions To Guarantee Progress

The protocol we just described does not guarantee progress – this is easily verified by running the protocol at a site s_3 that has two parents s_1 and s_2 which are incomparable in the DAG. A transaction T_1 with timestamp $(s_1, 1)$ will never be executed at site s_3 since $(s_2, j) < (s_1, 1)$, for all j. To ensure progress, we add an *epoch number* to each site's timestamp. The epoch number thus becomes a part of every transaction's timestamp. We use Definition 3.3 only for timestamps with the same epoch number. If timestamps TS_1 and TS_2 have different epoch numbers, then $TS_1 < TS_2$ if and only if the epoch number of TS_1 is smaller than the epoch number of TS_2. Clearly, this augmented definition of $<$ for timestamps with epoch numbers is also a linear order.

The protocol then executes secondary subtransactions in the order of their timestamps, with the new definition used to compare timestamps. Furthermore, the epoch number in a site's timestamp is always set to be equal to the epoch number of the most recent secondary subtransaction committed at the site. A transaction's timestamp has the same epoch number as that of the site when it committed. In order to ensure progress, the sources (sites with no parents) in the copy graph increment their epoch numbers periodically (with the same period). It follows from the above that the epoch number at a site is the minimum of the epoch numbers of the parents and that the epoch number at a site increases monotonically. This is sufficient to guarantee progress if there is a constant traffic on all edges of the copy graph. However, if there is no communication between a site and its child, then the child cannot advance its epoch number, since it waits until each incoming queue is not empty (see Section 3.2.3). Therefore, to guarantee progress, in the event that there has been no communication for a while, a site sends a "dummy" secondary subtransaction to its child – the dummy subtransaction has no updates but just pushes forward the site timestamp of the child. Thus, it follows that the epoch number increments eventually percolate to all sites.

Referring back to the example at the beginning of this section, it is now easy to see that T_1 will execute at site s_3 after some secondary subtransaction from s_2 with a bigger epoch number than $T_1's$ epoch number is at the head of the corresponding incoming queue.

4 The BackEdge Protocol

Both the DAG(WT) and the DAG(T) protocols, described in previous sections, require the copy graph to be acyclic to ensure serializability. The reason for this is that if the copy graph is permitted to contain cycles, then it may be impossible to ensure serializability in the lazy update propagation model (irrespective of the order in which updates are propagated), as illustrated by the following example.

Example 4.1 Consider a distributed system with two sites s_1 and s_2. Let site s_1 contain the primary copy of item a and a replica of b and let s_2 contain the primary copy of item b and a replica of a. Consider the following two transactions: T_1 at site s_1 that reads b and updates a and T_2 at site s_2 that reads a and updates b. Suppose both transactions execute at the two sites concurrently and commit. No matter which order we propagate updates of T_1 and T_2 to the other site, the

resulting schedule will always be non-serializable. The reason for this is that, due to read-write conflicts on a and b, at site s_1, T_1 will be serialized before T_2, while at site s_2, T_2 will be serialized before T_1. □

In this section, we propose the BackEdge protocol that ensures serializable schedules even though the copy graph contains cycles. It achieves this by adopting a hybrid approach, using eager propagation for some updates and lazy for the rest. The BackEdge protocol can be described as an extension of either the DAG(WT) or the DAG(T) protocol. Due to the lack of space, we will only discuss extensions to the DAG(WT) protocol here. ([BKRSS98] discusses extensions to the DAG(T) protocol.) Before, we delve into the extension, we need some terminology.

Let G be an arbitrary copy graph (which may contain cycles). Let B be a set of *backedges* in G. Recall that a set of edges in the copy graph are referred to as backedges, if their deletion breaks all cycles in the copy graph, that is, yields a DAG. Further, we will assume that B is a minimal set of backedges, that is, inserting any edge in B back into the resulting DAG causes a cycle in it. A set B for a graph can be computed easily using simple depth first search. We will discuss in Section 4.2 how the set B of backedges can be computed more cleverly. Let G_{dag} be the directed acyclic graph derived from G after deleting the edges in B from G.

4.1 Extending the DAG(WT) Protocol

Let T be the tree obtained by from G_{dag} that satisfies the required property (described in Section 2). If there exists a backedge from site s_i to s_j, then by the minimality of the set of backedges, it follows that there is a path from s_j to s_i in G_{dag}. Therefore, s_j is an ancestor of s_i in T, by the property of T. Thus, due to backedges, it is possible that a transaction needs to propagate updates to sites that are it's ancestors in T. Let T_i denote a primary subtransaction at site s_i. Let $S_1, \ldots, S_j$ denote the secondary subtransactions that execute at sites, $s_{i_1}, \ldots, s_{i_j}$, respectively, which are ancestors of s_i in T. We will refer to these secondary subtransactions as *backedge* subtransactions. Let s_{i_1} be the site that is farthest from s_i in T, s_{i_2} be the site that is the next farthest from s_i in T and so on. Then, the BackEdge protocol that extends the DAG(WT) protocol is as follows:

1. After T_i completes execution, the secondary subtransaction S_1 is directly sent to the site s_{i_1} to be executed (T_i continues to hold onto locks and has not yet committed).

2. After S_1 completes execution (it does not commit and holds on to its locks), it propagates the updates along the edges of the tree. Specifically, it forwards a "special" secondary subtransaction message containing its updates to its relevant children (note that the only child that is relevant is the one that is on the path from s_{i_1} to s_i in the tree). This special subtransaction is processed similar to other secondary subtransactions (e.g., FIFO order), except for the following difference. Until the secondary subtransaction reaches site s_i, none of the backedge subtransactions $S_2, \ldots, S_j$ that are executed along the path from s_{i_1} to s_i commit or release their locks. A site, when it receives the special secondary subtransaction from its parent, executes it and once it completes, the site forwards it to its relevant child without committing the substransaction.

3. After the special secondary subtransaction message from S_1 indicating that every backedge subtransaction has completed, is received at site s_i (and all secondary subtransactions received prior to it have been committed at s_i), T_i and subtransactions $S_1, \ldots, S_j$ are committed atomically (using a distributed commit protocol, e.g., two-phase commit) and locks held by them are released.

4. Once T_i commits at site s_i, the remaining secondary subtransactions for sites that are descendants of s_i in T are executed lazily following the DAG(WT) protocol (that is, by forwarding "normal" secondary subtransaction messages in which secondary subtransactions commit and release their locks before being forwarded to the relevant children).

Let us revisit Example 4.1 and trace the execution of the BackEdge protocol on that example. Transaction T_1 is allowed to commit at s_1, since it does not have a backedge subtransaction. Further, it is allowed to propagate it's update following the DAG(WT) protocol. However, transaction T_2, since it has a backedge subtransaction at site s_1, will hold onto its locks until the special subtransaction message from its subtransaction at site s_1 reaches site s_2. Meanwhile, T_1's subtransaction at s_2 will wait for T_2 to commit at s_2, since it needs the lock on item a that is held by T_2. Moreover, the special subtransaction message from s_1 for T_2 can be processed only after T_1's subtransaction commits (we process secondary subtransactions in FIFO order). Thus, since T_2 cannot commit until it receives the special subtransaction, there is a global deadlock involving T_2 and T_1's subtransaction. T_2 will be aborted (T_1's secondary subtransaction has to be completed before T_2 can commit and therefore it does not help if we abort T_1's secondary subtransaction).

Thus, the non-serializable execution of Example 4.1 is avoided.

Therefore, by holding onto locks for transaction T_i until the special subtransaction message is received at s_i, as in eager update propagation protocols, the backedge protocol ensures that T_i is committed only after transactions serialized before it at sites preceding s_i have committed. Thus, schedules stay serializable. Notice that locks for the backedge subtransactions $S_1, \ldots, S_j$ are continued to be held even after each of them completes execution – this enables us to abort these subtransactions in case there is a global deadlock and T_i needs to be aborted (as described above for Example 4.1). Holding these locks is not required for serializability, but only for atomicity.

Note that if the copy graph is a DAG, then there are no backedges and the BackEdge protocol reduces to the DAG(WT) protocol. Also, transactions for which there are no backedge subtransactions execute exactly as they would with the DAG(WT) protocol.

4.2 Minimizing the Effects of Backedges

Clearly, backedges are undesirable since they cause locks to be held by a transaction at multiple sites and for a longer duration when compared to the DAG protocols. Therefore, we need to minimize the number of times a transaction has to execute a secondary backedge subtransaction. In general, let there be weights associated with each edge in the copy graph which denotes the frequency with which an update has to be propagated along the edge. Then, to minimize the effects of backedges, we need to find a set of backedges B whose removal from G will give us a DAG and the sum of whose weights is minimum. If we assign a weight of one to each edge, then this is the feedback arc set problem which is NP-hard [GJ79].

Several approximation algorithms have been proposed for the weighted version of this problem [ENRS97, LMT90, ST97]. Any of these can be used to compute G_{dag} if the number of nodes in G is large.

5 Experimental Results

In order to evaluate the performance of our algorithms and to explore the dependency of our algorithms on values in the parameter space, we implemented a simpler variant of the BackEdge protocol (extension of DAG(WT)) and also a lazy version of the primary site locking algorithm (which is a variant of the lazy-master approach from [GHOS96]). Both were implemented using the DataBlitz product from Bell Labs as the underlying database [BLRSSS97]. The key point that differentiates DataBlitz from other commercial systems is that DataBlitz maps the entire database into the address space of the application process. A lock timeout mechanism is used to handle local as well as global deadlocks in our implementation. For our experiments, we set the timeout interval to be 50 millisec.

Our experiments were performed in a real-world setting involving 3 296 MHz Sun UltraSparc-2 machines running Solaris 2.6 and equipped with 256 MB of RAM. Since our performance study is based on a real implementation and *not* on a simulation study, we needed identical lightly loaded machines to run our experiments. That is the reason we had to restrict the number of machines in the study to 3. However, we ran multiple independent instances of DataBlitz on each machine in order to simulate multiple sites (one instance for each site). Thus, for experiments involving 9 sites, we would have 3 DataBlitz instances running on a single machine. The machines are on a 10 Mbit/sec ethernet network and all communication between programs running on a machine was performed using sockets and TCP as the transmission protocol.

5.1 Algorithms Implemented

We now describe details of our implementation of the primary site locking protocol and the simpler variant of the BackEdge protocol (extension of DAG(WT)). In this paper, we are primarily interested in distributed protocols. As a result, we do not consider here protocols that rely on a central site to ensure serializability (e.g., [ABKW98]).

Primary site locking protocol (PSL): In the PSL protocol, reads and updates by a transaction of items whose primary copies reside locally are handled at the site itself and the items are locked locally. However, reads of a replica are required to obtain a shared lock on the item at the primary site for the item. Also, the latest value of the item is shipped to the transaction along with the lock grant message. Update operations simply perform updates locally on the primary copy and *do not* propagate the updates to replicas. Thus, in our version of the PSL protocol, updates are propagated in the system lazily when the item is actually accessed by a remote site and there is no need to explicitly propagate updates to other sites. Therefore, all locks held by a transaction are released once it commits (even though the updates have not been propagated).

BackEdge protocol: In the variant of the BackEdge protocol (extension of DAG(WT)) that we implemented, instead of considering arbitrary trees, we consider the tree T (along whose edges updates are propagated) that is a chain (connect sites that are adjacent to each other in some total order of the sites consis-

tent with the DAG). Thus, our implementation is not as general and we expect the general implementation of the BackEdge protocol to outperform our implementation. Except for this difference, the remainder of the BackEdge protocol is identical to the description in Section 4.1.

5.2 Parameters Considered

We studied the performance of the two protocols for a wide range of parameter settings. Thus, we were able to characterize the parameter space under which each protocol can be expected to outperform the other. The parameters for each experiment and their default values are as shown in Table 1. Also, for parameters whose values were varied during experiments, we provide the range of values that were considered. The parameters that affect the performance of the schemes are the ones that control data distribution, transaction characteristics and system load. The default value of approximately 0.15 millisec for the network latency was not fixed by us, but this was the average communication latency we measured for our ethernet network. We now describe the data distribution and transaction generation schemes.

Data Distribution: Our data distribution algorithm assigns the primary copies of items uniformly across the m sites. The total number of distinct items (not counting replicas) is n. Thus, each site is the primary site for approximately n/m items. Of the primary copies assigned to a site, a fraction r of them are replicated. The remaining fraction $(1 - r)$ of them are not replicated and are thus local items at the site.

In our data distribution scheme, we utilize a total ordering on sites, $s_1, \ldots, s_m$ (that is consistent with the chain used by the BackEdge protocol to propagate updates), to distinguish between the DAG edges and the backedges. Thus, if a replica of an item at a site s_i is stored at a site s_j, and $j < i$, then this edge from s_i to s_j in the copy graph is treated as a backedge. For an item with primary copy at site s_i, replicas are assigned to the remaining sites according to parameters backedge probability b and site probability s. With probability b, all sites are considered as candidates for storing replicas of the item; and, with probability $(1 - b)$, replicas of the item are distributed only among sites that follow site s_i. Note that as b is increased, the number of backedges in the copy graph increases. Once the candidate sites for item replicas are determined, an item replica is assigned to a candidate site with probability s.

Transaction Generation: Each transaction is a sequence of 10 read or write operations and is run in the context of a thread. Each thread runs a sequence of 1000 transactions continuously one after another at a single site. The threads/site parameter specifies the number of threads that concurrently execute at a site and thus can be used to control the load in the system as well as the multiprogramming level. Obviously, more number of threads result in more contention within the system – we choose as the default value a multiprogramming level of 3 since we found that this generated a reasonable degree of contention for data items among the transactions. The read transaction probability is the probability that a transaction is labeled a read-only transaction (all the operations in such transactions are reads). If a transaction is not labeled a read-only transaction, the read operation probability is the fraction of operations within the transaction that are read operations. Fast access to an item is facilitated by a hash index on the item identifier, and appropriate shared/exclusive locks on the item are obtained when the item is accessed/updated.

5.3 Performance Results

We evaluated the two protocols based on mainly the following two performance metrics:

1. **Average Throughput:** This is the average of the transaction throughputs at each site. We only consider primary subtransactions for the throughput computation.

2. **Abort Rate:** This is the percentage of primary subtransactions that abort in the entire system. Due to lack of space, we do not show graphs of abort rate but report trends whereever appropriate.

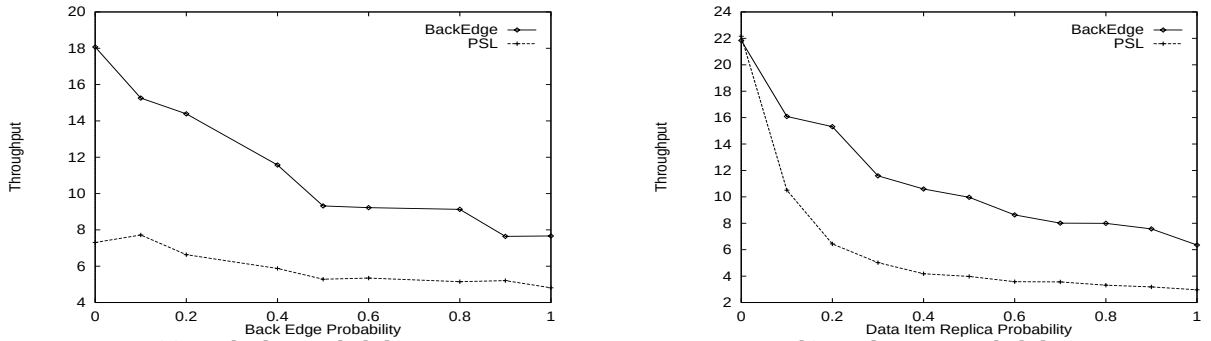
In the following subsections, we report the results of experiments in which we varied one parameter, while other parameters were set to their default values. In the default setting, shown in Table 1, we have 9 sites (3 per machine), 3 threads per site, 200 items overall, of which 20% are replicated at 50% of the sites, backedge probability b of 0.2, 50% read-only transactions and 70% of a transaction are read operations. Due to space constraints, we only present a subset of our experimental results – the full set of experiments can be found in [BKRSS98].

5.3.1 Backedge Probability

Figure 2(a) contains the graph for throughput, as b is varied from 0 to 1. When b is 0, there are no backedges and this is when the BackEdge protocol performs the best delivering almost thrice the throughput compared to the PSL protocol. The reason for this is that when $b = 0$, there are no backedge subtransactions and

Parameter	Symbol	Default Value	Range
Number of Sites	m	9	3 – 15
Number of Items	n	200	
Replication Probability	r	0.2	0 – 1
Site Probability	s	0.5	
Backedge Probability	b	0.2	0 – 1
Operations/Transaction		10	
Threads/Site		3	1 – 5
Transactions/Thread		1000	
Read Operation Probability		0.7	0 – 1
Read Transaction Probability		0.5	0 – 1
Network Latency		Approx 0.15 millisec	0.15 – 100 millisec
Deadlock Timeout Interval		50 millisec	

Table 1: Parameter Settings



(a) Backedge Probability (b) Replication Probability

Figure 2: Througput Results

each transaction executes locally, holding onto locks for a short duration. As a result, there are very few deadlocks and we observed that the abort rate was almost 0. As b is increased, an increasing number of transactions have backedge subtransactions and thus hold onto locks for longer intervals. The result is that the number of deadlocks in the system increases and so does the abort rate, thus decreasing the throughput.

Note that the PSL protocol is less affected as b is varied. There is a slight decrease in throughput and increase in abort rate as b is increased from 0 to 1. This is because, as b increases, so does the number of replicas in the system, and for the PSL protocol, as the number of replicas increases, the number of remote read operations increases as well, and so the performance becomes worse.

An interesting point to note is that even when $b = 1$, the BackEdge protocol performs better than the PSL protocol. Since in our default parameter setting, 50% of transactions are read-only transactions, and 70% of each transaction's operations are reads, even with $b = 1$, there are more remote read operations performed by the PSL protocol than secondary subtransactions generated by the BackEdge protocol, thus resulting in more message overhead.

For $b = 1$, it can be shown that for the BackEdge protocol, only 1 in 4 transactions require update propagation, while for the PSL protocol, each transaction performs about 4 remote reads on an average. Due to the higher communication overhead for the PSL protocol, it performs worse than the BackEdge protocol for $b = 1$.

5.3.2 Replication Probability

In Figure 2(b), we plot the throughput as the fraction of replicated items is increased from 0 to 1. The performance of both the BackEdge protocol as well as the PSL protocol deteriorates as the number of replicated items and thus the number of replicas in the system increases. This is expected since with more replicated items, a larger fraction of writes at a site are to replicated items and more read operations are directed to replicas. Also, as the number of replicated items increases, so does the number of backedges in the copy graph. Note the sudden drop in throughput from $r = 0$ to $r = 0.1$. This is due to the fact that when $r = 0$, none of the items are replicated and so

every transaction is a local transaction. Also, note that as expected both protocols have identical throughput at $r = 0$.

The performance of the BackEdge protocol is almost twice that of the PSL protocol for every r value except 0. This is because the number of replicas increases much more rapidly than the number of replicated items as r is increased (for instance, at $r = 1$, there are almost 500 replicas in the system). Thus, the probability of reading a replica (thus causing a remote read for PSL) is much higher than the probability of updating a primary copy of a replicated item. This coupled with the fact that 85% of all operations are read operations causes the PSL protocol's performance to be worse than that of the BackEdge protocol.

5.3.3 Extreme Parameter Settings

In this subsection, we consider two extreme parameter settings for backedge probability b – 0 and 1, in order to study the behaviour of the two protocols under extreme conditions. The values for most other parameters is set to their default values except for replication probability which is set to 0.5 and read transaction probability, which is set to 0. For both experiments, we vary read operation probability from 0 to 1.

Backedge Probability–0: The graphs for $b = 0$ as read operation probability is varied from 0 to 1 are shown in Figure 3(a). When read operation probability is 0, each transaction does only updates, and the PSL protocol performs better than the BackEdge protocol. This is because the PSL protocol does no remote communication and transactions execute completely locally. Even with the BackEdge protocol, transactions execute locally; however, they need to propagate updates to replicas and as a result, the BackEdge protocol needs to do more work than the PSL protocol and has inferior performance.

As the number of read operations is increased, the performance of the BackEdge protocol steadily increases as it needs to propagate fewer updates and the contention in the system decreases. The abort rates for transactions also decrease since there are fewer secondary subtransactions in the system and less contention as the number of reads increases. Finally, when transactions become completely read-only, the BackEdge protocol performs the best since transactions execute completely locally and don't have to propagate updates. The performance of the PSL protocol is somewhat more interesting. As the number of read operations is increased, the number of remote read operations increases and thus, the abort rate increases and the performance of the PSL protocol deteriorates until the read operation probability reaches about 0.5. However, beyond 0.5, the performance of the PSL protocol starts improving due to reduced contention until for a read operation probability of 1, there is no contention and the only additional overhead is that of reading remote replicas. Note that the throughput of the BackEdge protocol is more than 5 times that of the PSL protocol when read operation probability is 0.5.

Backedge Probability–1: Figure 3(b) shows the throughput as the read probability is varied. For $b = 1$, the PSL protocol behaves similarly to the case when $b = 0$. However, the performance of the BackEdge protocol lags the PSL protocol with respect to abort rate – this should be expected since there are a large number of backedge subtransactions and thus a large number of global deadlocks and aborts (since 50% of primary copies are replicated, almost every transaction generates a backedge subtransaction). The throughput of the BackEdge protocol is worse than PSL as long as the read probability is smaller than 0.3. Beyond a read probability of 0.3 (which is fairly small), however, the BackEdge protocol performs better despite the BackEdge probability being 1 and read transaction probability being 0.

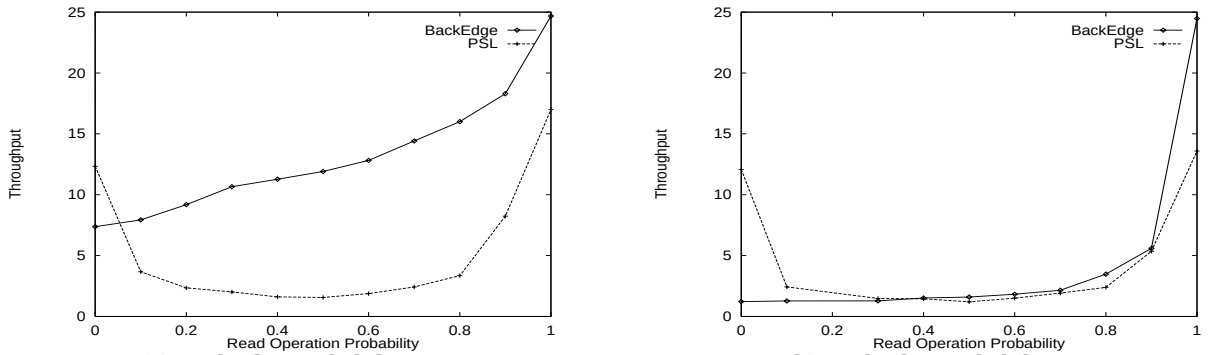
5.3.4 Other Performance Metrics

We also measured the average response time for committed transactions in each of our experiments. Due to space constraints, we do not report these in the paper. However, we found the response times to be related to the throughput and abort rate – in most cases (not always), we found the response times to be somewhat inversely related to the throughput, that is, the higher the throughput, the smaller the response times. Transaction response times for our experiments with the default parameter settings were approximately 180 millisec for the BackEdge and 260 millisec for the PSL protocol.

Another parameter that we do not report on is the time it takes a transaction's updates to propagate to all replicas in the BackEdge protocol. We did note that for our default parameter settings, update propagation via secondary subtransactions was extremely fast and in general took a few hundred millisec. As a result, we believe that recency of a site with the BackEdge protocols can be expected to be very good in practice.

6 Conclusions

In this paper, we proposed two new lazy update protocols, the DAG(WT) and the DAG(T) protocol that ensure serializability when the copy graph is a DAG. Thus, compared to prior work, we significantly expand the class of graphs for which lazy update



(a) BackEdge Probability-0 (b) BackEdge Probability-1

Figure 3: Throughput Results

protocols guarantee serializability. Further, in many real life situations, for example, a data warehousing environment, the copy graph is naturally a DAG. We also extended these protocols into the BackEdge protocol which ensures serializability for arbitrary copy graphs (that may contain cycles).

We implemented the BackEdge algorithm as well as a variant of the well-known primary site locking algorithm in the DataBlitz database product developed at Bell Labs. We conducted a detailed study of the relative performance of the two algorithms. The study revealed that unless transactions are update-intensive (more than 70% of all operations in a transaction are writes) and the copy graph contains a large number of backedges, the BackEdge algorithm consistently outperforms the primary site locking algorithm. For most practical environments, in which transactions are read-intensive and backedges are few, the BackEdge protocol can achieve speedups as high as five times compared to the primary site locking protocol. Thus, by implementing our protocols on top of off-the-shelf database systems, database designers can provide distributed applications with serializability and high performance in replicated environments.

References

[ABKW98] Todd Anderson, Yuri Breitbart, Henry F. Korth, and Avishai Wool. Replication, consistency and practicality: Are these mutually exclusive? In *Procs. of ACM SIGMOD International Conf. on Management of Data*, Seattle, WA, 1998.

[BK97] Yuri Breitbart and Henry F. Korth. Replication and consistency: Being lazy helps sometimes. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Tucson, Arizona, 1997.

[BKRSS98] Yuri Breitbart, Raghavan Komondoor, Rajeev Rastogi, S. Seshadri, and Avi Silberschatz. Update propagation algorithms for replicated database systems. Technical Report BL0112370-981028-11TM, Bell Labs, October 1998.

[BLRSSS97] Phil Bohannon, Daniel Lieuwen, Rajeev Rastogi, S. Seshadri, Avi Silberschatz, and S. Sudarshan. The architecture of the dali main memory storage manager. *Multimedia Tools and Applications*, 4(2), March 1997.

[CRR96] P. Chundi, D. J. Rosenkratz, and S. S. Ravi. Deferred updates and data placement in distributed databases. In *Proceedings of the Twelveth International Conference on Data Engineering*, New Orleans, Louisiana, 1996.

[ENRS97] Guy Even, Joseph (Seffi) Naor, Satish Rao, and Baruch Schieber. Fast approximate graph partitioning algorithms. In *Proceedings of the 8th ACM-SIAM Symposium on Discrete Algorithms*, New Orleans, Louisiana, 1997.

[GHKO81] J. Gray, P. Homan, H. Korth, and R. Obermack. A strawman analysis of the probability of wait and deadlock. Technical Report RJ2131, IBM San Jose Research Laboratory, 1981.

[GHOS96] J. Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *Proceedings of the ACM SIGMOD Conference*, Montreal, Quebec, Canada, 1996.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, San Francisco, California, 1979.

[LMT90] T. Leighton, F. Makedon, and S. Tragoudas. Approximation algorithms for vlsi partitioning problems. In *IEEE International Symposium on Circuits and Systems*, 1990.

[SK80] Avi Silberschtaz and Zvi Kedem. Consistency in hierarchical database systems. *Journal of the ACM*, 27(1), January 1980.

[ST97] H. D. Simon and S-H. Teng. How good is recursive bisection. *SIAM Journal of Scientific Computing*, 1997.