Second Order Snapshot-Log Relations: Supporting Multi-Directional Database Replication using Asynchronous Snapshot Replication

Yochai Ben-Chaim and Avigdor Gal
Technion – Israel Institute of Technology
{yochai@siglab, avigal@ie}.technion.ac.il

Abstract. Multi-directional asynchronous replication is a desired mechanism which allows updates to be performed locally at any site, and later propagated asynchronously to other sites. This paper proposes using second order snapshot-log relations as a mechanism for extending the use of single-directional asynchronous replication to multi-directional. The proposed mechanism is fully based on DBMS core tools and existing DBMS snapshot replication support, thus allowing a natural extension for systems that already support asynchronous snapshot replication. We have implemented and tested the proposed mechanism, showing results and terms of correctness.

1. Introduction

Distributed databases store data in multiple locations, allowing prompt response to users' needs. To increase retrieval speed and data availability, distributed databases may maintain *replicas*, copies of data, in several locations simultaneously. In the presence of replicas, user queries are computed using the replica that is *closest* to the user, in terms of either geographical proximity or network distance.

The database research community has put a great deal of effort into developing asynchronous protocols, in which replica updates are performed independently. The advantage of using asynchronous updates is that the transaction can be completed quickly and effectively at the local site, without needing to involve the whole system at once. The trouble with this concept, of course, is that if sites are updated independently, some replicas will contain stale data. Therefore, to be fully effective, asynchronous protocols must build ways to ensure timely updates at all sites (so that user queries are not answered with stale data) and to recover from replica inconsistencies due to local update failures.

Currently, the most common solution to the replica update problem is a process known as *snapshot view replication*. This process works as follows. The distributed database has a single main site to which updates are allowed. A site can serve as a main site for one sub-group of replicated relations, and a sub-site for others. In most cases, a single site serves as the main site for all replicated relations. Each relation maintains a snapshot-log, which holds all changes made to the relation. Replicas are read-only views, updated only by the DBMS using the snapshot-logs of the main site. Figure 1 illustrates the snapshot view replication as implemented within the Oracle system [3]. The snapshot-log relation *mlog$_R* is maintained by the DBMS, as it is used by the DBMS internal mechanisms to create the snapshot of *R*. At pre-defined periods (unrelated to the number and/or timing of changes), the DBMS in each sub-site (these are also known as *slaves*) uses the list of changes in the snapshot-log relation to update the replicated relation.

Using snapshot view replication simplifies the update process considerably, yet it may become too restrictive. For one thing, updates can be performed at a single site only, which serves as a single point of failure. Redundant servers provide only a partial solution, and often a costly one, as these can require a great deal of maintenance. The one site that is available for updates must be accessible from all other sites, both so these sites can be updated with replicated data, and so users on these sites can update data. Moreover, the load on this machine is likely to be high, putting stress on both the machine and the network connections.

An alternative to snapshot view replication would allow asynchronous multi-directional updates. In this case, updates of replicated relations would be allowed at any site, and replica updates would be performed asynchronously. As in the uni-directional model, security could be assured through techniques for controlling user permissions for updating relations.

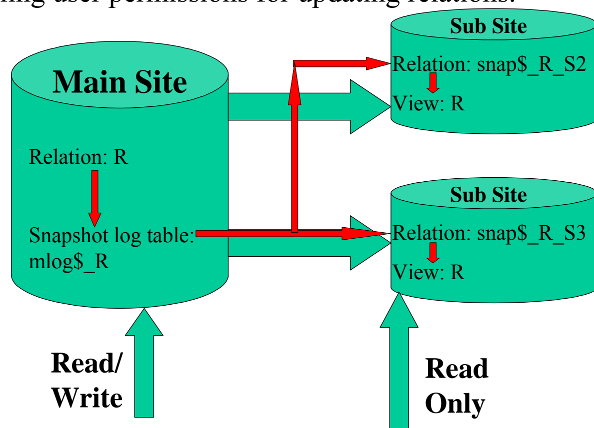

Fig. 1. Snapshot view replication in Oracle

Multi-directional replica updates have been previously suggested within synchronous protocols. Such updates are performed using the standard 2PC (2 Phase Commit) protocol [2], which is subject to deadlocks and network failures. Other synchronous protocols in the literature (*e.g.*, [19], [12]) attempt to improve on the 2PC model. Asynchronous bi-directional and multi-directional replica updates have also been suggested [4], [6]. However, these solutions have been based on tools such as vectors, objects and components, grouping of sites, and occasionally *special* reconciliation methods—none of them standard database components within the DBMS. Furthermore, these suggestions have never been implemented, and so have never undergone full performance analysis or comparison testing.

Vendor databases also offer multi-directional update replication (Oracle's update anywhere and SQL Server's merge replication). At this time, these extensions are criticized as costly (*e.g.*, Oracle's update anywhere is available only in the Enterprise edition which costs $20K more than the standard edition[1]), hard to design,[2] and bottleneck to performance (*e.g.*, Oracle's benchmark shows that update anywhere replication reduced a 200 update transactions per second machine down to 14 tps[3]).

This paper aims at providing a simple mechanism that can support multi-directional asynchronous database replication using asynchronous snapshot replication mechanism and standard database tools. The use of generic database tools — a feature we share with many others in database research — allows the DBMS to optimally manipulate the database resources and provides an encapsulated unified solution across platforms. The mechanism, dubbed *second order snapshot-log relations* is described in details in Section 3.

To highlight the capabilities of the proposed mechanism, we focus on the problem of preserving replica auto counter primary key attributes, as follows. An auto counter primary-key attribute is defined as a primary-key index attribute for the relation. Activating a triggered process on inserted tuples ensures the correct setting of values. Auto counter primary-key attributes are commonly accepted in DBMSs as offering the most reliable and efficient access to data in a relation ([15], [18]). The problem in correctly maintaining the auto counter attribute in an asynchronously replicated system is that changes to tuples can be performed at a site without knowledge of or regard to possible

[1] http://www.dbasupport.com/oracle/ora9i/ors.shtml
[2] http://asktom.oracle.com/pls/ask/f?p=4950:8:::::F4950_P8_DISPLAYID:14672061404704
[3] http://www.dbmsmag.com/9707d01.html

changes that have been made at other sites. For instance, a tuple may be added without the user being aware that at a different site a tuple has already been added and received the same auto counter attribute value. The solution we propose for this problem, using second order snapshot-log relations, is described in Section 4.

We advocate the use of the proposed mechanism as an easy-to-implement solution for cases where a full-fledged multi-directional replication mechanism is either too expensive or simply an over-kill for the application at hand. Using this mechanism, database designers can maintain their existing support of snapshot replication and enhance it to multi-directional replication mechanism when and where one is needed.

The remainder of this paper is organized as follows. Section 2 describes related work and lays the background for the proposed model. In sections 3 and 4, we present the suggested system architecture and update algorithm, and describe how asynchronous replication can be implemented within this architecture. We continue by providing the model's properties and complexity analysis in Section 5. Section 6 presents the simulation model and the result of our performance evaluation. We conclude in Section 7.

2. Related Work

Synchronized protocols consume many network and time resources, and are subject to deadlocks and extended locking periods while awaiting confirmation from all participants. Asynchronous models allow more rapid updates at one site at the cost of compromising information accuracy at other sites. Updates are performed at sites other than the original site at a later time, with the goal of mutual consistency. Taking into account both data currency and accessibility, the asynchronous model is still considered better than the synchronous model [4], [5], despite the risk of stale data at times, because of the high availability it offers.

The epidemic approach [8] has been suggested for asynchronous data management in distributed databases. According to this method, user operations are executed on a single replica, while in parallel, periodic pair-wise comparison of replicas is performed to detect and update obsolete copies. The epidemic approach requires increased overhead due to replica comparison, which is linear in the number of data items in the database. An alternative to this approach was suggested in [16], where the protocol detects whether update propagation between two replicas is needed at any given time, independent of the number of data items in the database. Should update propagation be needed, it is performed within a period of time linear in the number of data items to be copied. As a result, the number of data items that are frequently updated (and so need to be copied during update propagation) is much smaller than the total number of items in the database. However, the protocol is also based on asynchronous updates from a single updateable replica.

The Bengal database replication system [9] offers a stable mechanism for replication in wireless networks. As opposed to their work, we rely on database core functionality to exchange updates. Our novelty lies in the local architecture that allows a scalable and affordable transformation from uni-directional to multi-directional updates.

The mechanism proposed in this paper improves upon previously proposed bi-directional and multi-directional models [4] in two main respects. First, it is based on components and protocols that are already implemented in a vendor DBMS. Second, the proposed approach is progressive and non-blocking. By progressive we mean that the transaction’s coordinator always commits, sometimes with a group of other sites. Asynchronously, the update is later propagated to all other sites. By non-blocking we mean that each site can take unilateral decisions at each step of the algorithm. A reconciliation mechanism is responsible for bringing sites that cannot commit certain updates to mutual consistency, using history logs that are stored locally at each site. It is worth noting that if local replicas enforce the same set of integrity constraints, no reconciliation will be needed.

Standard software development methodologies, especially those that deal with information systems, stress the importance of using DBMS core tools as part of a strategy for dividing system implementation into separate and independent components ([14]). Using standard DBMS core tools allows full independent operation of the database, making the system scalable and enabling the replacement or upgrade of individual components. Among the advantages of using DBMS core tools, we can list ([7], [10]): improved flexibility of the system; reduced cost of system development and maintenance; ease of standards enforcement (including data naming, structure and formatting conventions); improved security and usage of security levels; and, finally, improved integrity, especially data integrity. The growing field of information engineering ([14]) also favors using the core features of the DBMS for maximizing system capabilities, as well as for efficient resource management.

In this paper we provide, as an example, the use of the proposed mechanism to maintain the uniqueness of auto counter primary keys. While solutions that involve designated offsets for various replicas [20] are common and may solve the uniqueness problem, it does not maintain the correct ordering of insertions to the database. Therefore, such solutions may not work whenever the application requires the tracking of the order of insertions through the primary key.

3. Second Order Snapshot-Log Relations

Consider the *base system* of asynchronous replication updates, as described earlier in Figure 1. A single primary site allows updates for each relation, which is later propagated to and updated at all other sub-sites. Figure 2 illustrates the proposed multi-directional extension of the uni-directional model case. In the multi-directional model, updates are available at all sites, and are later propagated to all other sites (relations *snap$_R_Si* where $i \in \{1, 2 \ldots n\}$). In this model there is no distinction between a master site and slave sites. All sites allow updates, and all receive propagated updates from all other sites. To achieve this, the relation *R* is defined as a read-write relation at all sites.

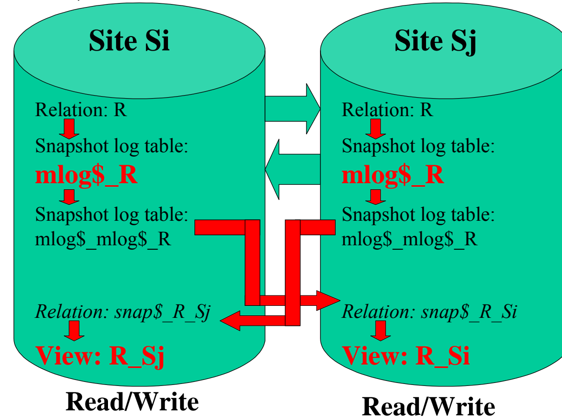

Fig. 2. Multi-Directional Asynchronous Replication Model

We next describe our solution to enhance existing asynchronous snapshot replication mechanism to handle multi-directional updates. As a first step, we create a snapshot-log relation *mlog$_R* (dubbed *1st order snapshot-log relation*) holding all changes made to *R*. This snapshot-log relation is the same one used in uni-directional snapshot-based asynchronous replication. Next, we create an additional snapshot-log relation (dubbed *2nd order snapshot-log relation*), based on the *mlog$_R* relation and holding all changes made to *mlog$_R* (and indirectly to *R*). The 2nd order snapshot-log relation is used to replicate the *mlog$_R* relation to other sites (denoted *mlog$_mlog$_R* in Figure 2), as shown by the arrows in Figure 2.

In the third step, we create a snapshot of all instances of *mlog$_R* relations from any site to all other sites (denoted *snap$_R_Sj* and *snap$_R_Si* in Figure 2). That is, the relation *snap$_R_Sj* in site *Si* (hence also the view *R_Sj*) is (periodically) identical to the snapshot log relation *mlog$_R* on site *Sj*, and vice versa.

The necessity of using a 2nd order snapshot-log relation *mlog$_mlog$_R* is illustrated in Figure 1, stems from the need to define the relation *R* to be of type read/write at all sites. Recall that in the uni-directional model, updates to the base relation are only available at a single site, while at all other sites the relation is read-only. Therefore, the system alone has the authorization to update *R* using *mlog$_R*. This is no longer the case for the multi-directional model. Here, *R* can be freely updated by the user and the system replica maintenance can no longer guarantee consistency. Therefore, to ensure consistency, each site should have (1) an automatic mechanism to receive updates from all other sites, and (2) a mechanism to propagate these updates to *R*. The former is achieved by using *R_Si* as the read-only relation, capturing updates to *R* in site *Si*. The latter is achieved by adding a trigger-based algorithm that propagates these updates to local *R* relations.

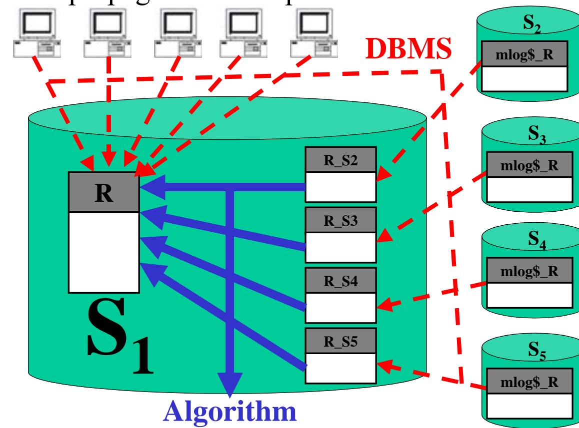

Fig. 3. Algorithm extension of DBMS core standard features

To recap, Figure 3 illustrates an example of the extension over the DBMS core standard features, involving five sites. DBMS core tools allow users to perform read/write operations on relation *R* in site S_1. The DBMS core tools also automatically create and update the snapshot log relations named *mlog$_R* in sites S_2, S_3, S_4, and S_5, and create a **read-only** view of those relations (marked *R_S2, R_S3, R_S4* and *R_S5* respectively) at site S_1. The algorithm provides the tools for correctly updating the base relation *R* from the list of changes received from the other sites.

Using this architecture, each site S_i, of the *n* sites $S_1, S_2 \ldots S_n$, holds *n*-1 snapshots (*snap$_R_Si*) of each of the *mlog$_R* relations from the other *n*-1 sites. In accordance with the snapshot replication mechanism, only the changes in the 2nd order snapshot-log relation are actually replicated from a given site to each other site. Each site's DBMS, upon receiving the changes in the 2nd order snapshot-log relation, uses these changes to construct the correct view of the snapshot-log relation. This occurs automatically by means of the snapshot replication mechanism.

An important feature of the 2nd order snapshot-log relation is its ability to allow each site to construct a list of the changes according to a timestamp. This allows the site to employ deterministic logic to consolidate changes made at the local site ([11]) with those made at other sites, sequentially applying the changes in the local relation. We assume that all sites have a common datetime clock, based on known synchronization algorithms (*e.g.*, [13]). The common datetime clock ensures a chronological update ordering. It is worth noting that the common datetime clock does not mean that the updates between the different sites need to be synchronized. On the contrary, the algorithm is independent of the order in which sites receive updates from other sites, and the algorithm does not enforce synchronization of the update process.

4. Update Propagation

We term the process by which updates from a server are propagated to other servers *update propagation*. To describe the update propagation process, let us consider the case of a single change made on relation R at site S_i (denoted $S_i.R$), and let us show how this change is propagated for execution on each of the other local relations $S_j.R$ for each $j \neq i$. We start with an overview of the process (Section 4.1), followed by an example (Section 4.2). The complete description of the process is given in [1].

4.1. Process Overview

As a change made to relation $S_i.R$ is committed, the DBMS snapshot replication mechanism creates a tuple in the local *mlog$_R* (snapshot-log) relation at site S_i containing the information on the change made. A tuple created in *mlog$_R* reflects a change to $S_i.R$. This operation is accompanied by the creation, by the DBMS, of a new tuple to the relation *mlog$_mlog$_R*, containing information on changes to *mlog$_R*. The information in these tuples includes the type of change, the values of each attribute before and after the change (where applicable), and the timestamp of the change. A tuple representing a change of type *insert* contains all the inserted tuple's attribute values. A tuple representing a change of type *delete* contains only the auto counter attribute value. A tuple representing a change of type *update* contains the auto counter attribute value as well as the attribute values of the updated attributes, both from before and after the change. It should be noted that snapshot-log relations (both 1st and 2nd order) are append-only relations, as all change types in the base relation generate new tuples in the snapshot-log relation.

The DBMS common uni-directional snapshot replication model uses the *mlog$_mlog$_R* relation to replicate the *mlog$_R* relation from S_i to all other sites S_j such that $j \neq i$. This method ensures that the change in *mlog$_R* relation from S_i is propagated to all of the other n–1 sites, among them site S_j.

As site S_j receives a change in the *mlog$_R* replicated relation from S_i, a triggered operation is run, verifying whether this change has already been executed on $S_j.R$. This verification process is conducted according to the type of change and the change's unique timestamp. It compares the local copy of changes to R (*mlog$_R*) and the local relation holding the list of changes on R from the remote site S_i – *snap$_R_Si*. To speed up the retrieval process from these relations, indices are added to the snapshot-log relation and the replicated relations on the timestamp of the change.

Should the verification process yields that a change has already been executed on $S_j.R$, then site S_j disregards it. Otherwise, site S_j performs the change on the local base relation, following the sequence of timestamps received from site S_i to maintain the order among changes. The timestamp serves for later propagating this change to all other sites, as well as for inserting the tuple with the correctly calculated auto counter attribute value. The auto counter attribute value is calculated to reflect the relative datetime value of the inserted tuple. Therefore, if additional tuples are inserted after this tuple, its auto counter attribute value will be altered to reflect the addition. Changes imposed on the auto counter attribute values of tuples should not be propagated to other sites, as the triggered operations at those sites will make the same adjustments upon receiving the change in relation R. According to the algorithm, the order in which the site receives the updates performed at other sites is immaterial (see Section 5 for algorithm properties).

The triggered operation is executed as an atomic transaction at the local site, ensuring that all updates from a given site are processed before those from a different site. Tuples from any given site are handled in the order of their timestamp. Handling the changes according to the timestamp assures that updates or deletions of tuples received from other sites are related one-to-one to tuples that have already been added in the local site. The local site can only receive changes of type *update* or *delete* that are one-to-one related to tuples for which the local site has already received a change of type *insert*. However, it is possible for an update to be received at a site where the relevant tuple has been deleted. In this case, it may appear that the later update implicitly suggests re-insertion of a tuple

accompanied by an update. However, the proposed algorithm assumes that deletion of tuples has a higher priority over later updates to the same tuples.

4.2. Triggered Operation Algorithm Example

The algorithm is triggered upon the arrival of new tuples to the replicated copies of snapshot-log relations. The triggered operation is executed as an atomic transaction at the local site. The full algorithm contains a lot of bookkeeping and is provided in [1]. Similar approaches are available in the literature using other data structures (*e.g.*, version vectors [17]). This section provides an example run, describing how sites are updated.

As an example, consider a network environment consisting of three sites: S_1, S_2, and S_3. We assume that at time t, all three sites are synchronized. We consider a single relation R, having at timestamp t a maximum auto-counter attribute value of x.

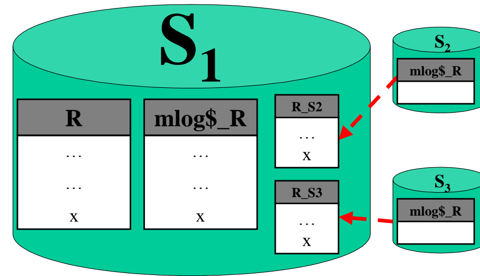

Fig. 4. Relations at site S_1

Figure 4 illustrates the relations at site S_1. The site contains the base relation R, the snapshot log relation *mlog$_R*, and the replicated views *R_S2* and *R_S3*, which are copies of the snapshot log relations from sites S_2 and S_3 respectively.

Accordingly, site S_2 contains the base relation R, the snapshot log relation *mlog$_R*, and the replicated views *R_S1* and *R_S3,* and site S_3 contains the relations R, *mlog$_R*, *R_S1* and *R_S2*.

We consider the following list of changes executed at timestamps $t < t_1 < \ldots < t_6$:

t_1 - a tuple is inserted at site S_1 and receives an auto counter attribute value of $x+1$.

t_2 - the tuple with an auto counter attribute value of $x+1$ is updated at site S_1.

t_3 - a tuple is inserted at site S_2 and receives an auto counter attribute value of $x+1$ (without knowledge that a different tuple has already received this value at site S_1).

t_4 - the tuple with an auto counter attribute value of $x+1$ is updated at site S_2.

t_5 - the tuple with an auto counter attribute value of x is deleted at site S_2.

t_6 - the tuple with an auto counter attribute value of x is updated at site S_1.

Next, the list of changes *mlog$_R* from site S_2 is replicated to site S_1.

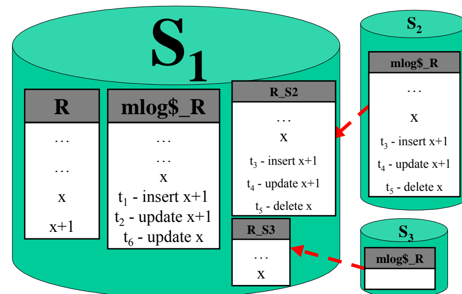

Fig. 5. After snapshot log relations are replicated

Figure 5 illustrates the relations at site S_1 after the snapshot log relation *mlog$_R* is replicated from site S_2 to site S_1. A triggered operation at site S_1 on new tuples in relation *R_S2* activates the

algorithm. The algorithm checks the new tuples and calculates their net effect on the local base relation *R*.

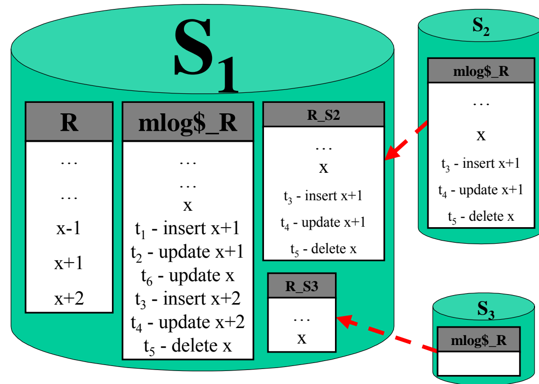

Fig. 6. Site S_1 at the end of the triggered operation

The algorithm checks the list of changes from *R_S2* against the list of local changes from relation *mlog$_R*. According to the order of timestamps, the algorithm concludes that the tuple added at t_3 should receive an auto counter attribute value of *x+2* and not *x+1*. It then concludes that the tuple updated at t_4 is the tuple with the new auto counter attribute value of *x+2*. Next, the algorithm deletes the tuple with an auto counter attribute value of *x*. Figure 6 illustrates site S_1 at the end of the triggered operation.

As site S_2 receives the replicated tuples in relation *R_S1*, the triggered operation concludes that the auto counter attribute value of the local tuple with an auto counter attribute value of *x+1* is updated to *x+2*, the new tuple that arrived from site S_1 is inserted (and then updated) with an auto counter attribute value of *x+1*, and the update of the tuple with an auto counter attribute value of *x* is disregarded, as this tuple has been deleted.

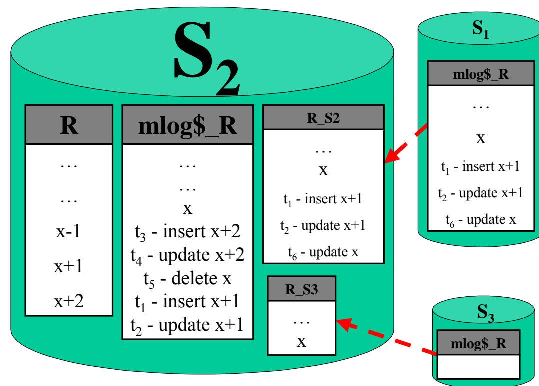

Fig. 7. Site S_2 at the end of the triggered operation

Figure 7 illustrates site S_2 at the end of the triggered operation on inserted tuples from relation *R_S1*. It is worth noting that while the list of activities in both sites is not identical (S_1 contains the update in t_6 while S_2 does not) the net effect is the same, since the tuple with the key *x* has been deleted.

Site S_3 receives the changes either from site S_1 and then from site S_2, or vise versa. If the former, the actions at site S_3 will be similar to those that occurred at site S_1. If the changes from site S_2 are received first, the actions at site S_3 will be similar to those that occurred at site S_2. At the end of the triggered operations, and in the absence of other update operations, all three sites will contain the same tuples in the base relation *R*.

5. Main Properties of the Model

The main properties of the model (formally proven in [1]) are:

- **Soundness** - Let R be a relation with local copies $S_1.R$, $S_2.R$ … $S_n.R$. Assume that a change u, identified by a change type t_u and timestamp value dt_u, was performed on relation $S_i.R$, $1 \leq i \leq n$. There exists a site S_j (possibly $S_j = S_i$) at which u is a user-triggered change.
- **Completeness** - Consider a relation R with local copies $S_1.R$, $S_2.R$ … $S_n.R$. Assume that a change u, identified by a change type t_u and timestamp value dt_u, was initiated by a user on relation $S_i.R$, $1 \leq i \leq n$. For each S_j ($j \neq i$, $1 \leq j \leq n$) there exists a timestamp t in which $u \in S_j.snap\$_R$.
- **Termination** – Assume a timestamp t_1 after which users initiate no further changes. There exists a timestamp t_2 ($t_2 > t_1$) such that for each $t_3 \geq t_2$, $snap\$_R_{t3} = snap\$_R_{t2}$.

  We define two tuples t_1 and t_2 in relation R on two different sites S_i and S_j as *identical* ($S_i.R.t_1 = S_j.R.t_2$) if the following two conditions hold: a) the two tuples have the same creation timestamp; and b) the two tuples have the same values for all attributes, with the possible exception of the auto counter key attribute.

  We define two relations $S_i.R$ and $S_j.R$ to be *consistent* if for every tuple t_1 in $S_i.R$ there exists an identical tuple t_2 in $S_j.R$ such that $S_i.R.t_1.acfv = S_j.R.t_2.acfv$ (where *acfv* is the auto counter value) and vise versa.
- **Correctness** – Assume that changes at all sites have stopped at timestamp t_1. There exists a timestamp t_2 at which $S_i.R$ and $S_j.R$ are consistent.

The complexity analysis of the update propagation mechanism describes the cost in terms of storage space and number of triggered operations. In terms of storage space: assume n sites, and assume that m changes were made on the local relation $S_i.R$ and that the relation $S_i.R$ holds k tuples. The uni-directional snapshot replication model requires a total of $n \times k+m$ tuples at the n sites. Assuming $O(m)=O(k)$, we arrive at a cost linear both in n and in m.

In the multi-directional model, the storage space needed sums to $n \times k + n \times m$ tuples. Again, assuming $O(m)=O(k)$, we arrive at a cost linear in n and in m. Clearly, the multi-directional model requires additional $(n-1) \times m$ more tuples than the uni-directional model.

In terms of number of triggered operations: in the uni-directional model each triggered operation performs a single change at each of the n sites, or $O(m)$ for m update operations.

In the multi-directional model, each change results in a triggered operation at each of the ($n-1$) sites. Each change is then propagated again to each of the other $n-1$ sites, thus resulting in $O(mn)$ triggered operations for m update operations. The second time a change arrives at a site, it is identified as a change that has already been performed at the site, and is not performed again. According to the algorithm, each triggered operation may affect additional tuples. In the worst case, each triggered operation affects $O(n)$ tuples, and we arrive at a total of $O(mn^2)$, which has a similar theoretical performance to that of the benchmark results of Oracle's update anywhere. Optimization to this process, such as the use of partitions according to the timestamp values to speed up searches, is beyond the scope of this paper.

6. Simulation Model

We have implemented the proposed mechanism and tested it using a simulation with three different sites and one base relation replicated at all sites. Each site had a Windows 2000 Server machine, single CPU, with Oracle Server version 8.0.5 that was configured to allow replication operations. The proposed multi-directional model was implemented using standard DBMS tools (Oracle DBMS and PL SQL tools).

The network we used is a standard TCP-IP Microsoft Windows network, and a LAN type network configuration. The database connection definitions were based on TCP-IP addresses.

The simulation shows the relative performance of the multi-directional protocol vis-à-vis the uni-directional model in scalability and practical functional values of the algorithm. We measured:

- CPU usage of the servers.

- The amount of data transferred over the network.
- The average wait-time for up-to-date data in the nearest server when performing a query.

6.1. Experiment Setup

Each run began with an empty relation at each site. This is not a prerequisite of the algorithm, yet served to simplify the calculations. During the test runs, the relations at each site were populated. All the changes were of type *insert* to simplify the simulation. We expect to achieve similar performance with a combination of update operations as well.

In what follows, we use the following terminology:
Network Usage: the total number of bytes transferred.
Reconciliation: the process by which each server sends updates performed on the local relation to all other servers, and performs updates received from all other servers. The reconciliation process ends when no updates remain to be propagated.
Low Stress: server operates below full capacity.
High Stress: server is required to perform more operations per second than it can handle.
Database Server Availability: A server is considered *available* when its CPU usage falls below 100% utilization. Requests to a server whose CPU is being fully utilized are queued for later handling. A server's availability is measured in the amount and percentage of time in which the server's CPU is utilized at less than 100%.

6.2. Experiment Results and Performance Analysis

The CPU usage experiments show that for low-stress situations—$X/9$ and $X/3$, where X is the number of requests per second that brings the server to 100% utilization (approximately 32 in the experiment setup used) — the multi- and uni- directional models exhibit almost identical availability. For high-stress situations, X and $3X$, there are significant differences, and the multi-directional protocol allows for periods of time in which the server CPUs are utilized at less than 100%. This is because in the multi-directional model the load is divided among several servers, whereas in the uni-directional model a single master server performs all updates.

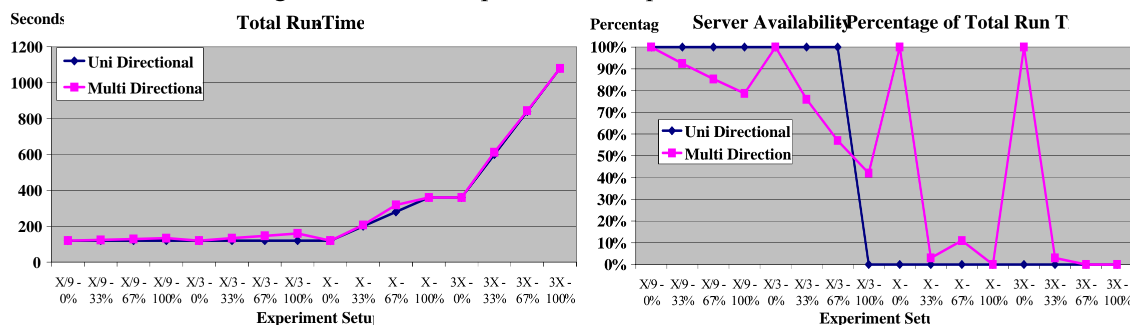

Fig. 8. Server availability and overall run time until the reconciliation process ended for different stress levels (relative to X) and degrees to which update requests make up of the total requests

Figure 8 illustrates our results for the master server in the uni-directional model, and a representative server for the multi-directional model. The figure shows server availability as a percentage of run time (on the right), and overall run time (on the left), for 16 experiments at different stress levels. As can be seen, the servers in the multi-directional model show better availability under high stress conditions, while the uni-directional model shows slightly shorter run times (despite the order of magnitude theoretical difference stated in Section 5).

6.3. Network Topology

The experiments were performed over a LAN network where all servers and all the users were connected over the same LAN, so that network traffic was not such that would cause bottlenecks or

high network stress. Under these conditions, our experimental results show that the network traffic imposed by the multi-directional model is roughly n times (where n is the number of sites) greater than that needed for the uni-directional model.

The experiments for this paper showed that network usage is not a factor over a LAN, as the server usage limit is exceeded far before the network bandwidth is exhausted. An additional parameter over networks is the delay time. Over a LAN, the delay time is negligible.

Taking into account both network usage and delay time, the multi-directional model shows an incremental advantage over the uni-directional model in the transaction and propagation processing procedures. In the uni-directional model, each transaction is completed only after being committed on the master server. In the multi-directional model, on the other hand, each transaction initiated by a user is usually completed within the local sub-network, and so is not subject to delay times or network bottlenecks.

Using the results of the experiments performed over a LAN network, we were able to calculate the average wait-time for up-to-date data in the nearest server for each user. Looking again at Figure 8, recall that we used 16 different experimental setups, with four stress levels (*X*/9, *X*/3, *X,* and 3*X*) and, within each stress level, four degrees to which update requests make up of the total requests (0%, 33%, 67%, and 100%). Cases where updates made up all requests or none are irrelevant for this experiment and so can be discarded, leaving us with 8 experimental setups.

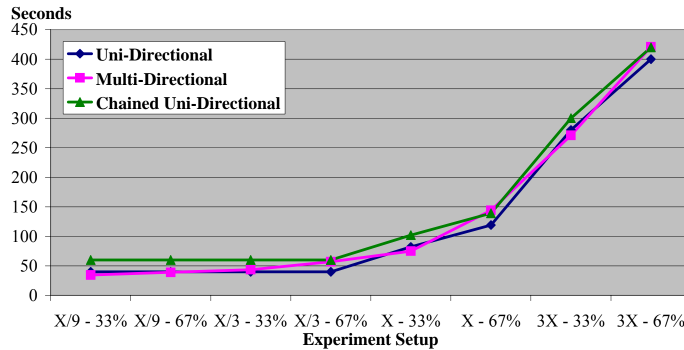

Fig. 9. Wait-Time for Up-To-Date Data Comparison between the Multi- and Uni- Directional Models over a WAN network

Figure 9 shows the average wait-time (in seconds) for the most up-to-date data under uni- and multi-directional protocols for these 8 setups. Over a LAN network, the uni-directional model shows slightly shorter wait-times than does the multi-directional model.

Figure 9 represents the wait-time for up-to-date data comparison between the multi- and uni-directional models, as expected over a WAN environment with an average delay time of five seconds. As can be seen, roughly half of the experiment setups show a better average wait-time for the multi-directional model over the uni-directional model. For the chain variation, the uni-directional model shows longer wait-times than the multi-directional model in almost all the experiments. In general, the multi-directional model will show better wait-times than the uni-directional model as the delay over a WAN network increases.

7. Conclusion

In this paper we propose a multi-directional extension to the uni-directional asynchronous replication model, synchronizing updates in the presence of auto counter primary keys. We have introduced second order snapshot log relations as the mechanism and presented an algorithm to maintain multi-directional asynchronous replication, while conserving the relations' auto counter attribute correct value (for use as a primary-key attribute). The model makes it possible to build a

distributed system that is updated asynchronously (update margin intervals are scalable to system needs) without having to compromise on the use of auto counter primary key attributes or on the ability to alter each relation at each site, at a cost measured only in disk space and minor DBMS resources. An important attribute of this mechanism is that it is based fully on DBMS core tools, thus allowing a natural extension for systems that already support the uni-directional asynchronous update model.

We have implemented and experimented with the proposed mechanism in a simulated environment. Experimental results show the terms under which better performance and server availability is expected from the multi-directional model. Additionally, over a WAN network, the uni-directional model experiences a much slower update process as compared to the multi-directional model. In the multi-directional model, the delay times are only evident in the asynchronous update process. User-initiated update transactions are usually performed over a LAN. In the uni-directional model, the user suffers slower response times when performing update transactions, as these usually need to be committed over a WAN.

References

[1] Removed for the sake of double-blind review process.

[2] Bernstein, P. A., Hadzilacos, V. and Goodman N. (1987), "Concurrency Control and Recovery in Database Systems", Addison-Wesley.

[3] Bobrowski, S. and Smith G. (Primary Authors - 1997), "Oracle8 Replication, Release 8.0, Part No. A58245-01", Oracle Corporation.

[4] Ceri, S., Houtsma, M. A. W., Keller, A. M. and Samarati, P. (1992), "Achieving Incremental Consistency among Autonomous Replicated Databases", *Proceedings of the IFIP WG 2.6 Database Semantics Conference on Interoperable Database Systems (DS-5)*, pp. 223-237.

[5] Ceri, S., Houtsma, M. A. W., Keller, A. M. and Samarati, P. (1995), "Independent Updates and Incremental Agreement in Replicated Databases", *Distributed and Parallel Databases*, 3(3), pp. 225-246.

[6] Chang, T. P. and Hull, R. (1995), "Using Witness Generators to Support Bi-directional Update Between Object-Based Databases", *Proceedings of the fourteenth Symposium on Principles of Database Systems (PODS)*, pp. 196-207.

[7] Dadam, P. (2000), "On the Design, Implementation, and Maintenance of Enterprise-wide Transactional Workflow Applications for Advanced Environments: Challenges and Open Issues", position paper, http://www-adele.imag.fr/IPTW/IPTW/Papers/.

[8] Demers, A., Greene, D., Hauser, C., Irish, W., Larson, J., Shenker, S., Sturgis, H., Swinerhart, D. and Terry, D. (1987), "Epidemic Algorithms for Replicated Database Maintenance", *Proceedings of the 6th Symposium on Principles of Distributed Computing (PODS)*, pp. 1-12.

[9] Ekenstam, T., Matheny, C., Reihner, P., and Popek, G.J. (2001), "The Bengal Database Replication System", Distributed and Parallel Databases, 9(3), pp. 187-210.

[10] Elmasri, R. and Navathe, S. (2000), "Fundamentals of Database Systems (3rd Edition)", Addison-Wesley.

[11] Goldring, R. (1995), "Things Every Update Replication Customer Should Know", *Proceedings of the International Conference on Management of Data (SIGMOD)*, pp. 439-440.

[12] Hsu, M. and Silberschatz, A. (1991), "Unilateral Commit: A New Paradigm for Reliable Distributed Transaction Processing", *Proceedings of the Seventh International Conference on Data Engineering (ICDE)*, pp. 286-293.

[13] Lamport, L. (1990), "Concurrent Reading and Writing of Clocks", ACM Trans. On Computer Systems, vol. 8, pp. 305-310.

[14] Martin, J. (1989), "Information Engineering: Introduction", Prentice Hall Professional Technical Reference.

[15] Melonfire, I. (2002), "PHP Application Development With ADODB", Developer Shed Network Site, www.devshed.com.

[16] Rabinovich, M., Gehani, N. H., Kononov, A. (1996), "Scalable Update Propagation in Epidemic Replicated Databases", *Proceedings of the 5th International Conference on Extending Database Technology: Advances in Database Technology (EDBT)*, pp. 207-222.

[17] Ratner, D., Reiher, P., and Popek, G. (1997), "Dynamic Version Vector Maintenance", Computer Science Department: University of California, Los Angeles, 1997.

[18] Reed, J. (2003), "Carbon User Manager Rdbms Usage", Sapient.

[19] Singhal, M. (1990), "Update Transport: A New Technique for Update Synchronization in Replicated Database Systems", *IEEE Transactions on Software Engineering (TSE)*, 16(12), pp. 1325-1336.

[20] Soparkar, N. and Silberschatz, A. (1990), "Data-value Partitioning and Virtual Messages," *Proceedings of 9th A CM SIGA CT-SIGMOD-SIGART Symposium on Principles of Database Systems,* Nashville, TN.