E. Pacitti, E. Simon, and R. Melo

# Update Propagation Strategies to Improve Data Freshness in Lazy Master Schemes

submitted to ICDCS'98

1998

# Update Propagation Strategies to Improve Data Freshness in Lazy Master Schemes *

Esther Pacitti[1,3,†], Eric Simon[1], Rubens Melo[2]

November 5, 1997

[1] Projet Rodin, INRIA, Rocquencourt, France
[2] Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio), Brazil
[3] Núcleo de Computação Eletrônica da UFRJ (NCE-UFRJ), Brazil
e-mail: firstname.lastname@inria.fr, rubens@inf.puc-rio.br

**Abstract**

Many distributed database applications need to replicate data to improve data availability and query response time. The two-phase-commit protocol guarantees mutual consistency of replicated data but does not provide good performance. *Lazy replication* has been used as an alternative solution in several types of applications on-line financial transactions and telecommunication systems. In this case, mutual consistency is relaxed and the concept of freshness is used to measure the deviation between replica copies. In this paper we present a framework for lazy replication and focus on a special replication scheme called *lazy master*. In this scheme the most common update propagation strategy is based on the deferred propagation: changes on a primary copy are first committed at the master node, and afterwards the secondary copy is updated in a separate transaction at the slave node. We propose update propagation strategies that use immediate propagation: updates to a primary copy are propagated towards a slave node as soon as they are detected at the master node without waiting for the commitment of the update transaction. We propose an update propagation architecture, study the behavior of our strategies and show that they may improve data freshness with respect to the deferred approach.

*This work is partially supported by Esprit project EP 22469DWQ "Foundations for Data Warehouse Quality".

†The author was supported by Conselho Nacional de Pesquisa (CNPq) as a Phd candidate of PUC-Rio in a colaboration program with Projet Rodin.

## 1 Introduction

Replicated data facilitate the improvement of query performance and increase data availability in database applications. Replicated data involves a set of replica copies that are stored on distinct nodes of an interconnected system. A central problem of several database applications with real-time constraints, such as telecommunication systems [GHK$^+$97], on-line financial transactions [Sha97] is to guarantee a high level of freshness of replicated data. For example, in a global trading system distributed over a wide-area network, it is crucial to have fast access to exchange rates from any trader location. This can be achieved by replicating exchange rate data. However, a change to a rate by a trader at a location must be propagated as soon as possible to all other locations to refresh replicated data. With a *two-phase-commit* protocol (henceforth 2PC), each time an *update transaction* updates a replica of an object at some node, all the other replica copies of this object get updated within the same transaction, thereby guaranteeing that all replica copies are *mutually consistent*. However, this solution does not scale up because it requires a round trip message between the node where the update of a replica is initiated and every other node with a replica of the same data. Therefore, performance degrades as the number of nodes increases. Futhermore, 2PC is a blocking protocol in the case of network or node failures [PHN87].

An alternative is to use a *deferred propagation* approach in a *lazy* replication scheme [Lad90, Gol95] which works as follows: each time a transaction updates a replica at some node, it is first committed at that node, and then every other replica copy of the same object is updated in a separate *refresh transaction*. In the example of Figure 1, after a transaction updates $R$ at node 1, two refresh transactions are generated to update $R$ at nodes 2 and 3. This scheme relaxes the mutual consistency property assured by 2PC. However, the interval of time between the execution of the original update transaction and the corresponding refresh transactions may be significant due to the time needed to update propagate the refresh transactions. The notion of *freshness* indicates the number of updates that are not reflected by a given replica but have nevertheless been performed on the other replica copies (the smaller this number, the fresher is the replica).

In this paper we address the problem of freshness in lazy replication schemes. We present a lazy master framework and architecture and propose the use of *immediate propagation* to improve freshness: updates are propagated as soon as they are detected. Specifically, we propose two update propagation strategies: *immediate-immediate* and *immediate-wait*. With *immediate-immediate*, the refresh transaction is executed as soon as it is received and with *immediate-wait* the refresh transaction is executed only when its *commit* is received. We present experimental results that demonstrate the freshness performance of these strategies with respect to a *deferred* one that

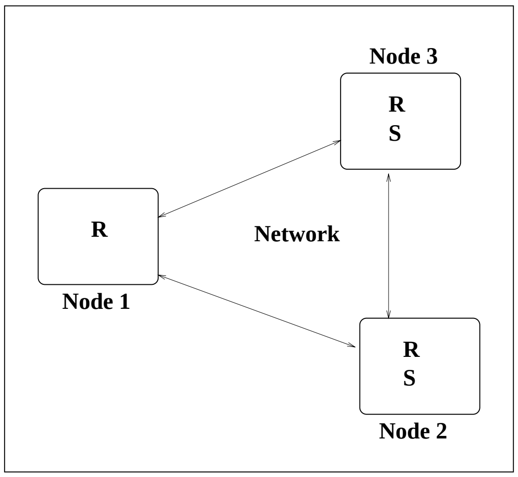

Figure 1: Replicated Database System

is implemented by some commercial relational database systems. The results indicate that for short transactions the deferred approach performs almost as well as *immediate-immediate* and *immediate-wait*. The strategies exhibit different freshness results when long transactions are introduced. In these cases, our strategies show much better results and the *immediate-immediate* strategy provides the best freshness of data. For some workloads the freshness is twice that provided by the deferred strategy. On the other hand, *immediate-wait* only improves freshness when the update transaction arrival rate at the master is bursty. The downside of the *immediate-immediate* strategy is the increase of query response time due to the lock holding time. However, we show that using a multiversion protocol at the slave node, this drawback can be drastically reduced without a significant loss of freshness.

The remainder of this paper is structured as follows. Section 2 introduces the replication framework we use, our vocabulary and useful definitions. Section 3 presents the update propagation architecture used to implement our strategies. Section 4 describes the details of the deferred strategy we consider and our immediate ones. Section 5 presents our performance evaluation. In Section 6 we relate our work to other existing work. Finally, Section 7 concludes.

## 2 Preliminaries

The next few paragraphs presents the lazy replication framework we propose and describes the the scheme we are interested in. We assume familiarity with transactional concepts that are well documented in [PHN87].

We characterize a lazy replication scheme using four basic parameters: ownership, propagation, refreshment and configuration. The *ownership* parameter [GHOS96] defines the permissions for updating replica copies. If a replica copy is updatable it is called a *primary* copy, otherwise it is called a *secondary* copy. In the rest of this paper, we focus on a specific case called *lazy master* replication scheme. In this scheme there is a single primary copy of an object, all its other replicas being secondary copies. The node that stores the primary copy of an object is called a *master* for this object, while the nodes that store its secondary copies are called *slaves* [GHOS96]. The *propagation* parameter defines *when* the updates to a replica must be propagated towards the nodes storing the other replicas of the same object. The *refreshment* parameter defines the *scheduling* of the refresh transactions. If a refresh transaction is executed as soon as it is received by some node, the strategy is said to be *immediate*. The juxtaposition of the propagation and refreshment parameters determines a specific update propagation strategy. For instance, a *deferred-immediate* update propagation strategy has a deferred propagation and an immediate refreshment. The *configuration* parameter characterizes the nodes and the network.

We use $R$ to denote a primary copy and to $r$ to denote a secondary copy of relation $R$. Since we are interested in freshness improvement, we focus on a *one master-one slave* configuration to present most of our algorithms. Each update transaction $T_i$ that updates $R$ has a corresponding refresh transaction, $RT_i$, that is executed on each slave node that stores $r$. Each query $Q_i$ that reads $r$ is a read-only transaction. The number of read operations performed by a $Q_i$ defines its *size*. Each refresh transaction $RT_i$ is composed of the sequence of write operations $< w_1, ..., w_n >$, ignoring reads, performed by $T_i$ to update $R$. The number of writes carried out by a $T_i$ or $RT_i$ defines its size.

We assume that messages are exchanged among the nodes of the replicated system through a reliable communication network that checks for errors, and loss and duplication of messages. Furthermore, messages are received in the same order they are sent (*order preserving*).

## 3 Architecture

The underlying idea of our architecture is to maintain the autonomy of each node. This means that neither the local transaction management protocols nor query processing are changed to support a lazy master replication scheme. Each node, whether master or slave, is equipped with three components, in addition to the database system. The first component is the replication module, which itself consists of a *Log Monitoring, Propagator and Receiver*. The second component is the *Refresher*, which provides different qualities of service by implementing different refreshment strategies to update secondary

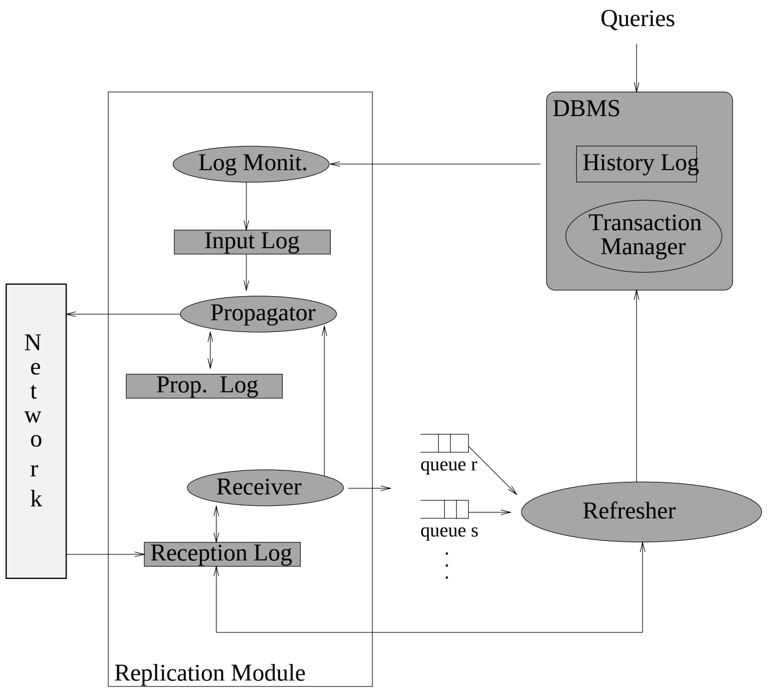

Figure 2: Architecture of a node: square boxes represent persistent data repositories and oval boxes represent system components

copies. The last component, which is the *network interface*, that is used to propagate and receive messages does not appear in Figure 2 and are not discussed in this paper. We detail the functionality of the replication module and the refresher in the following:

**Log Monitoring:** The Log Monitoring implements *log sniffing* [SKS86, KR87, Moi96], which is a procedure used to extract the changes to a primary copy by continously reading the contents of a local history log (noted $H$). When the log monitoring process finds a write operation on $R$, it reads the corresponding log record from $H$ and writes it into a stable storage called *input log* that is used by the Propagator.

**Receiver:** It implements message reception at the slave node. Messages coming from different masters are received through a network interface that stores them in a *reception log*. The receiver reads messages from the *reception log* and stores them in *dynamic queues*. The contents of these queues form the input to the *Refresher*. A slave node is able to detect its master node failure using *time-out* procedures via the network interface. As soon as a failure is detected, it is signaled to the refresher. On the other hand, during a slave recovery, the receiver uses the *reception log* to check out the last received message. Afterwards, the receiver signals its recovery to its master node using its propagator.

**Propagator:** It implements message propagation that carries log records, issued by the Log Monitoring, or reception recovery signals issued by the receiver. Both types of messages are written in the *input log*. The propagator continously reads the records of the *input log* and propagates messages through the network interface. A master node is able to detect failures of its slave nodes using the network interface. As soon as a failure is detected, the propagator deactivates propagation towards the failed node. Propagation is only reactivated when the master receives the recovery signal coming from the failed node. On the other hand, during a master recovery, the propagator uses the *propagation log* to restart propagation.

**Refresher:** The refresher process implements refreshment strategies that define *when* (e.g. immediately, periodically) and in *which order* refresh transactions are executed. It reads the contents of the *dynamic queues*, one for each secondary copy, and following some refreshment strategy, it executes refresh transactions to update secondary copies. A refresh transaction execution is performed by submitting each write operation to the *local transaction manager*.

## 4 Update Propagation Strategies

We consider three update propagation strategies: *deferred_immediate*, that is based on the common approach used in existing lazy master replication schemes [Moi96], and our *immediate_immediate* and *immediate_wait* strategies. In this section, we present them with respect to our architecture. We first present the deferred and immediate propagation strategies that establish the basis for other strategies. We also show how we deal with failures. In the last subsection, we propose an optimization of the *immediate-wait* strategy.

### 4.1 Propagation

The Log Monitoring process reads log records from $H$ in the same order they were written to preserve *serializability*. It is important to mention that the algorithm used to manage $H$ in the local database system is orthogonal to our strategies and have no impact in the way they function. The propagator process also reads log records from the input log in the same order they were written to propagate them in serial order. The input log stores log records and possible signals issued by the slave receiver. Each log record stored

```
Propagator
input: input log
output: messages sent to slave nodes
variables:
    $o$: a record read from the input log
    $m_i$: a message that carries $o$
    $M_i$: message that carries a sequence of $o$ associated with some $T_i$
    begin
        repeat
            read(input log,$o$);
            if propagation = **immediate**
                    *propagate* ($m_i$);
            else /* propagation = **deferred** */
                if $o = commit$ for $T_i$
                    propagate($M_i$);
                if $o = abort$ for $T_i$
                    discard($M_i$);
                if $o = write$
                    add $o$ to the corresponding sequence $M_i$;
        for ever;
    end.
```

Figure 3: Propagator Algorithm

in $H$ and in the input log carries the information necessary to perform an operation, that is the following atributes (see [GR93]):

**<timestamp, primary_id,tuple_id, field_id,operation,new_value>**

Each node implements a logical clock inside the local transaction manager. The *timestamp* consists of the transaction's execution start time (using local clock). The master identification, that in our case is the *primary_id*, identifies the primary copy $R$ that was updated at the master node. Tuples are identified by their primary keys. In addition, the updated field within a tuple is identified by *field_id*. Next, *operation* identifies the type of operation (update, delete, insert, abort and commit) performed by a $T_i$. In case of update operations, *new_value* contains the new value of the field being updated. When there is no ambiguity, we sometime use the term operation in place of log record.

The propagator implements the algorithm given in Figure 3. For clarity, the handling of failures, which is the focus of a next subsection, is not represented in this algorithm. With an *immediate* propagation, each write operation, $w_i$, of $R$, by transaction $T_i$ is read from the input log and afterwards, a function *propagate* forwards a message $m_i$ containing the input log record to each slave holding a copy $r$. It is important to note that concurrent update transactions at the master produce an interleaved sequence of write operations in $H$ for different update transactions. However, write operations are propagated in the same order as they were written in the input log to preserve the master serial execution order. Update transaction's *aborts* and *commits* at a master are also detected by the log monitoring process.

With *deferred* propagation, the sequence $< w_1, w_2, ...commit >$ of operations on $R$ done by transaction $T_i$ (henceforth, called the refresh transaction $RT_i$), is packaged within a single message, noted $M_i$, that is propagated to each slave holding some $r$, after reading $T_i$'s *commit* from the *input log*. In case of an *abort* of $T_i$, all the corresponding $M_i$ under construction are discarded using the *discard* function.

## 4.2 Reception and Refreshment

Each slave receiver reads the messages $M_i$ in their reception order from the reception log. In addition, each receiver checks for the master identification *primary_id* field to select in which queue to store the log record(s) that each message carries. [1] We assume that when a message carries a sequence, this sequence is stored as a single record in the queue.

[1] with *deferred-immediate*, a message carries a whole sequence of log records, otherwise, a message carries a single log record.

To update a secondary copy $r$, the refresher continously reads the dynamic queues seeking for new incoming records. We assume , for the purpose of this paper, that the ordering of refresh transactions is irrelevant. With *deferred-immediate*, the refresher reads a sequence $< w_1, w_2, ..., commit >$ from a queue and subsequently submits it as a refresh transaction to the local transaction manager. Note that the effect of the serial execution order of the update transactions $T_1, T_2..., T_n$ performed at the master is preserved at the slave because the corresponding refresh transactions $RT_1, RT_2..., RT_n$ are performed in the same order.

With *immediate-immediate*, each time a write operation is read from a queue it is subsequently submitted to the local transaction manager as part of some refresh transaction. Here again, for the same reasons as before, the effect of the serial execution order of the update transactions performed at the master is preserved. When an abort operation of a $T_i$ is read by the refresher it is also submitted to the local transaction manager to abort $RT_i$.

With *immediate-wait*, the sequence $< w_1, w_2, ...w_k, commit >$ of operations in queue $q_r$, associated with a refresh transaction $RT_i$ is stored into a dynamic auxiliary structure, noted $RV_i$, called a *reception vector* This vector has one entry per write operation. When $RT_i$'s *commit* is read by the refresher in queue $q_r$, the sequence of operations of $RV_i$ is read and submitted to the transaction manager using the *apply* function. Afterwards, $RT_i$'s *commit* is submitted. The period of time during which the refresher waits for reading $RT_i$'s *commit* is called the *wait period.* Figure 4 summarizes the algorithm executed by the refresher for a given queue $q_r$ using immediate-wait.

For all three strategies, when a refresh transaction $RT_i$ is committed, the refresher marks all the records in the reception log that carry a $M_i$ (deferred-immediate) or $m_j$ message (immediate-immediate and immediate-wait) as *processed.*

## 4.3 Dealing with Failures

Node recovery is treated through the execution of three asynchronous procedures: *propagation*, *reception* and *refreshment* recovery. On master propagation recovery, the propagator checks for the last propagated message $M_i$ or $m_i$ in the *propagation log* and restarts by reading the next one, $M_{i+1}$ or $m_{i+1}$, from the *input log.* During a slave reception recovery, for each master, the receiver checks for the last received message, $M_i$ or $m_i$, in the *reception log* and asks each master (through the propagator) to restart propagation starting from $M_{i+1}$ or $m_{i+1}$. In *refreshment recovery,* all messages received but not *processed* are read from the *reception log* and are placed in the correct queue.

```
Immediate_Wait
input: a queue $q_r$
output: submit refreshment transactions
variables:
    $o$: the contents of $m_i$ stored in $q_r$
    $RV_i$: dynamic vector of write operations for $RT_i$
    begin
        repeat
            read($q_r$,$o$);
            if $o$ corresponds to a new $RT_i$
                    Create a new $RV_i$;
            if ($o \neq$ commit) and ($o \neq$ abort))
                    add($RV_i$, $o$);
            if $o =$ commit for $T_i$
                    apply($RV_i$);
                    submit($o$);
            if $o =$ abort
                    discard($RV_i$);
        for ever.
    end.
```

Figure 4: Immediate-Wait Algorithm for queue $q_r$

## 4.4 Optimization for Immediate-Wait

We present an extension of the *immediate-wait* strategy that takes adavantage of the wait period to optimize the execution of refresh transactions. In many applications, such as a data warehousing applications [FMS97], replicas are used in a slave node as the operand relations of materialized views. Essentially, the replication mechanism is used to propagate incremental changes from a data source (i.e. a master node) to a data warehouse (i.e. a slave node). Thus, the transactions that refresh a replica in the slave are then propagated to refresh the materialized views, thereby entailing the execution of new *view-refresh* transactions.

We supose that every refresh transaction to $r$ generates a *view-refresh* transaction to a materialized view $V_r$ defined with $r$; which is the case for some data warehousing applications, aslo called Data Stores [FMS97].

In fact, incremental view refreshment algorithms take as input the *net changes* to their operand relations [GMS93]. Our first optimization is to compute, for each $RT_i$, the next change replica during the wait period. This is performed using the $RV_i$. For instance, if a sequence of write to a same data item occurs in $RV_i$, only the last one is kept.

The second optimization is to start the *view-refresh* transactions generated by a $RT_i$ as soon as the $RT_i$ is started, directly using the $RV_i$, that is, without waiting for the $RT_i$ *commit* execution at the slave. Futhermore, the computation of the changes to be made to a view $V_r$ can be computed from the $RV_i$ as soon as records are entered into $RV_i$.

## 5 Performance Evaluation

In this section, we present our performance study to understand the tradeoffs of freshness and performance for the *deferred-immediate*, *immediate-immediate* and *immediate-wait* strategies in a one master - one slave configuration.

### 5.1 Definition of Freshness

Using the concepts introduced in previous sections, the notion of freshness is formalized as follows. We shall assume that all transactions have the same size, that is the same number of write operations. Using this assumption of uniformity, the freshness of $r$ at time $t$ is the difference between the number of committed update transactions on $R$, noted $n(R)$, and the number of committed refresh transactions on $r$, noted $n(r)$, at time $t$:

$$freshness(t, r) = n(R) - n(r)$$

## 5.2 Simulation Environment

The main factors that influence the freshness of replicas are: (i) $T_i$'s execution time, reflected by the time spent to log monitor $T_i$ in $H$, (ii) $RT_i$'s propagation time, that is the time needed to propagate the necessary messages for $RT_i$, from a master to its slave, and (iii) $RT_i$'s execution time. Consequently, our performance study only focuses on the components of a node architecture that determine these three factors.

It is clear that freshness can be improved by increasing the processing capability of a slave node or the speed of the network media. However, our experiments concentrate on the impact of update propagation strategies to the freshness of replicas.

The simulation model is an open queueing model involving four asynchronous processes. The model is implemented on a Sun Solaris using pipes for inter process communication. Figure 5 portrays the complete simulation environment.

The first process, called *master*, simulates the behaviour of the log monitoring and propagator for a single database connection. The *density* of an update transaction $T_i$ is the average interval of time (noted by $\epsilon$) between write operations in an update transaction as it is reflected in $H$. If, on average, $\epsilon \geq c$, where $c$ is a predefined system parameter, then $T_i$ is said to be *sparse*; otherwise, $T_i$ is said to be *dense*. We focus on *dense* update transactions. In addition, we vary $T_i$'s arrival rate distribution in $H$ (denoted by $\lambda_t$). We consider writes as *update* operations; each operation updates the same attribute (noted *atr*) of a different tuple. Each propagation message ($M_i$ or $m_i$) is written in order into a *pipe* $p_1$ and is later read by the *network* process.

The *network* process simulates the communication network. The input to this process is the propagation messages written by the *master* process. The network is modeled as an FCFS server. Network delay is calculated by $\delta + t$, where $\delta$ is the network delay introduced to propagate each message and $t$ is the *on-wire* transmission time. In general, $\delta$ is considered to be non significant and $t$ is calculated by dividing the message size by the network bandwidth [CFLS91]. In our experiments, we use short message transmission time (noted $t_{short}$) of 300ms, which represents the time needed to propagate a single log record. In addition, we consider that the time spent to transmit a sequence of log records is linearly proportional to the number of log records it carries. The network overhead delay to propagate each message is modeled by the system overhead to read and write from pipes. The *Total propagation time* (noted $t_p$) is the time spent to propagate the log records involved in a refresh transaction $RT_i$. Thus, with immediate propagation, $t_p = n \times (\delta + t_{short})$, while with deferred propagation, $t_p = (\delta + n \times t_{short})$. *Network contention* occurs when $\delta$ increases due to the increase of traffic on the network. In this situation, the delay introduced by $\delta$ may impact the

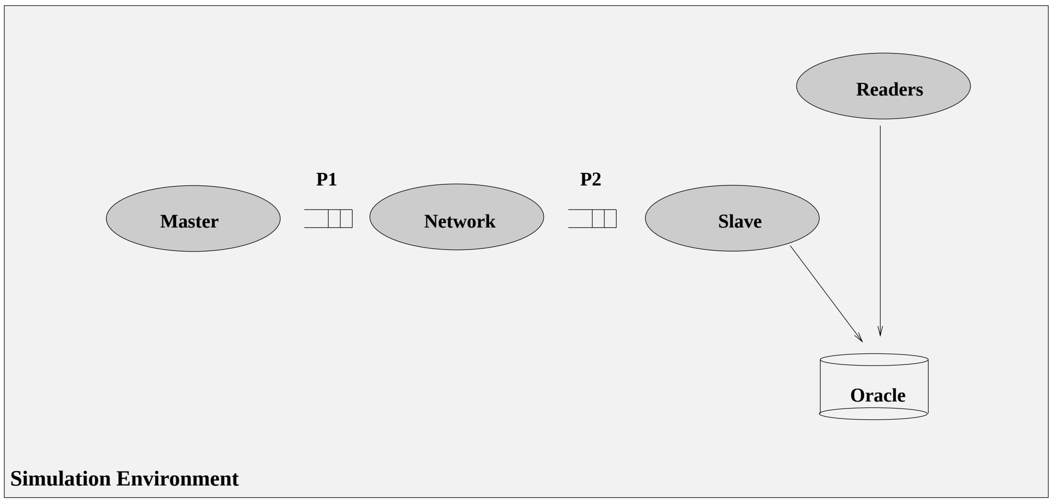

Figure 5: Simulator

total propagation time, specially with immediate propagation. The output of the *network* process are the contents of the messages that are written, in order, into pipe $p_2$ that are later read by the *slave* process.

The third process, called *slave*, simulates the refresher at the slave node. The *refreshment time* is the time spent to execute an $RT_i$ and the *update propagation time* is defined as the time delay between the commitment of $RT_i$ at the slave and the commitment of its corresponding $T_i$ at the master. Refresh transaction execution is performed on top of an Oracle 7.3 system using C/SQL. Each operation corresponds to an UPDATE command that is submitted to the server for execution. A replica $r$ is a relation taken from the AS3AP benchmark [Gra91] and is populated with 20,000 tuples.

Finally, the forth process, called *readers*, implements query processing on the slave. The query expression we use is *Select atr from r where atr* $\geq c_1$ *and atr* $\leq c_2$, where $c_1$ and $c_2$ are fixed. Query arrival rate distribution (noted $\lambda_q$) is defined as a workload called *low*. Query size (noted $Q$ size) is fixed to 5.

Using strict two phase locking (henceforth, S2PL) as the underlying concurrency control protocol may increase a query's response time when the query conflicts with a refresh transaction. To improve query response time, we consider the use of a multiversion two phase locking (henceforth, MV2PL) and examine its impact on freshness. MV2PL exploits versions to increase concurrency between transactions. The principle is that queries read committed versions of data items, whereas update transactions write new ones. It is important to notice that using MV2PL, queries never conflict with refresh transactions.

| *Parameters* | *Values* |
|---|---|
| $\epsilon$ | Exponential $(mean = 100ms)$ |
| $\lambda_t$ | Exponential: *low* $(mean = 10s)$,*bursty* $(mean = 200ms)$ |
| $\lambda_q$ | Exponential: *low* $(mean = 15s)$ |
| $Q$ size | 5 |
| $RT$'s sizes | 5,50 |
| Conflicts | 50% |
| Protocols | S2PL, MV2PL |
| $t_{short}$ (1 write) | 300ms |
| $ltr$ | 0,30,60,100 |

Table 1: Performance Model

To measure freshness and compare the impact of using S2PL and MV2PL we fix a 50% conflict. This means that the refresh transactions update 50% of the tuples that are to be read by a query. We simulate S2PL by using the *select* command followed by *for update*. Since refresh transaction contains only write operations MV2PL is easily realized by using Oracle multi-version consistency model [Bob96].

We define two types of update transactions. Small update transactions have a size 5 (i.e., 5 write operations), while long transactions have a size 50. To understand the behavior of each strategy in the presence of short and long transactions we define four scenarios. Each scenario determines a parameter called *long transaction ratio* (noted $ltr$). In scenario 1, $ltr = 0$ (only short update transactions are executed), in scenario 2, $ltr = 30$ (30 % of the executed update transactions are long), in scenario 3, $ltr = 60$ (60 % of the executed update transactions are long), and in scenario 4, $ltr = 100$ (all executed update transactions are long). The parameters of the performance model are summarized in Table 1.

The results are average values obtained for the execution of 40 update transactions.

## Experiment 1

The goal of this experiment is to analyze the average freshness and query response time for a *low* update transaction arrival rate at the master.

As despicted in Figure 6, when $ltr = 0$, freshness = 0.3, i.e., replicas are quite fresh, with the three strategies. The reason is that on average, $\lambda_t \simeq t_p$, that is the time interval between the execution of a $T_i$ and a subsequent $T_{i+1}$ is sufficiently high to enable completion of $RT_i$'s update propagation before the commitment of $T_{i+1}$. However, for higher $ltr$ values, $\lambda_t < t_p$ for the three strategies. Thus, during $RT_i$'s update propagation, some transac-

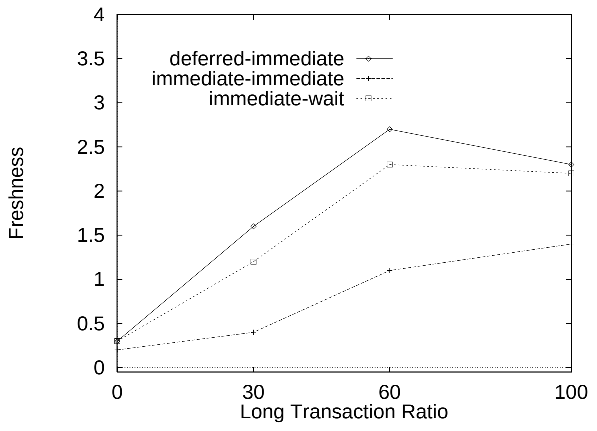

Figure 6: Low Workload - Freshness

tions $T_{i+1}...T_{i+n}$ may be committed, thereby increasing the value of freshness. For all $ltr$ values, the freshness values obtained with *deferred-immediate* and *immediate-wait* are close because the refreshment time is near equal for these two strategies. Futhermore, the total propagation times are also close since there is no network contention.

With *immediate-immediate*, refreshment time is larger compared to the other strategies because the time interval between the submission of $w_j$ and $w_{j+1}$ of $RT_i$ depends on $t_{short}$, $\delta$ and $\epsilon$, slowing down refreshment time. However, *immediate-immediate* update propagation time is smaller, compared with *immediate-wait* and *deferred-immediate* ones, because propagation and refreshment are done simultaneously. That is, $RT_i$ execution starts after the reception of the first write. Therefore, *immediate-immediate* is the strategy that always presents the lowest freshness results. For all strategies, the freshness results do not vary linearly with $ltr$ since we are mixing transaction sizes and our freshness measure is based on transaction size.

The delay introduced when using S2PL in a conflict does not impact significantly refreshment time because the query size is small but query response times may be increased. This is observed especially with the *immediate-immediate* strategy because the refreshment time increases due to the overlapping of propagation and refreshment. However, the chance of conflicts is reduced because $\lambda_t \simeq \lambda_q$. That is the reason why query response times are not seriously affected when $ltr = 30$ (see Figure 7). However, with $ltr = 60$ and $ltr = 100$, lock holding times are longer due to the transaction size, causing the increase in query response times. Figure 7 shows a situation (when $ltr = 100$) where response time may be doubled compared with *immediate-wait*. We only show the *immediate-wait* curve since response times for the *deferred-immediate* strategy are very close. When using MV2PL, query re-

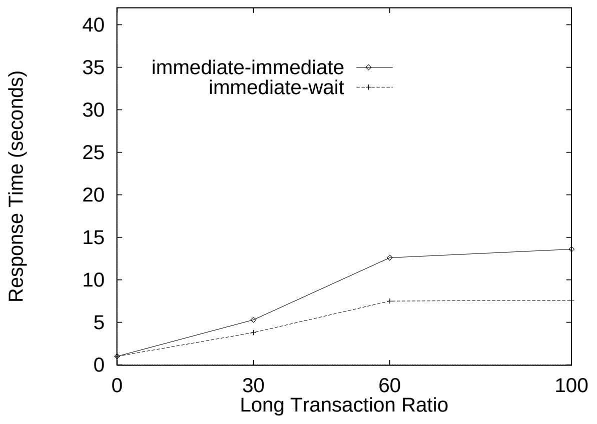

Figure 7: Low Workload - Response Time

sponse time for the three strategies in all cases is reduced to an average of 1.2sec.

**Experiment 2**

The goal of this experiment is to show the values of both freshness and query response times for the three strategies for a high workload that we call *bursty*.

As despicted in Figure 8 when $ltr = 0$ (only short transactions) freshness is impacted because on average, $\lambda_t < t_p$. Therefore, during $RT_i$ update propagation $T_{i+1}, T_{i+2}...T_{i+n}$ may be commited. It is important to note that *deferred-immediate* presents lower freshness results compared to *immediate-wait* in bursty workloads because $\delta$ increases sufficiently to increase the *immediate* total propagation time. Therefore, the total propagation time of a short refresh transaction using *deferred* propagation may be less than the total propagation time using *immediate* propagation. On the other hand, even with network contention, *immediate-immediate* shows lower results compared with both *deferred-immediate* and *immediate-wait*, because refreshment begins after the reception of the first write.

When long update transactions are executed, freshness results are higher because $t_p$ increases and $\lambda_t << t_p$. Notice that *immediate-wait* begins to improve and becomes better than *deferred-immediate* when $ltr = 30$, $ltr = 60$ and $ltr = 100$. This is because $q_r$ is quickly filled with a large number of operations such that when the refresher reads $q_r$ seeking for a new $RT_i$, the all or almost all of $RT_i$ may be already stored in $q_r$. In this case, the additional *wait period* of *immediate-wait* may be reduced or even become 0. This is cleary seen when $ltr = 100$ (all refresh transactions have the same size). Figure 10 shows a freshness snapshot for a sequence of query executions

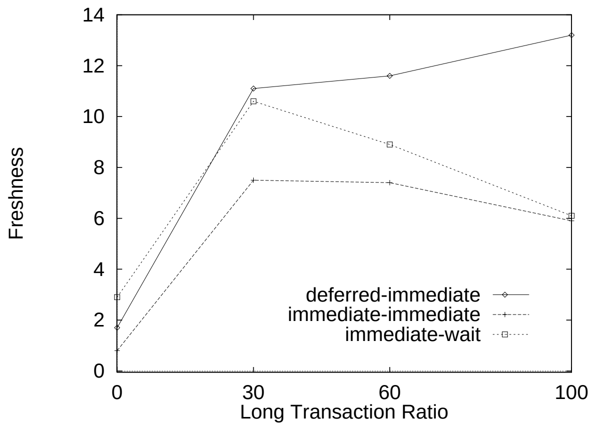

Figure 8: Bursty Workload - Freshness

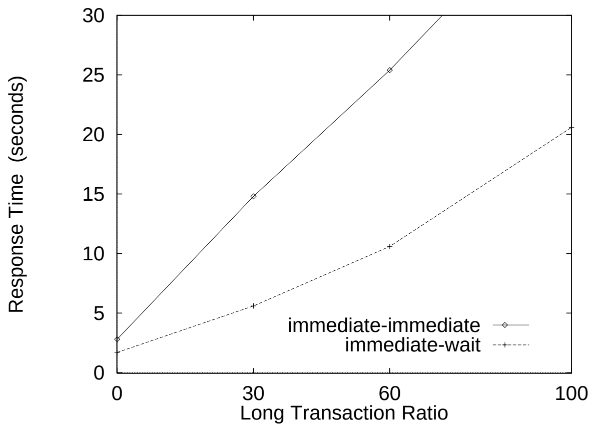

Figure 9: Bursty Workload - Response Time

when $ltr = 100$.

Notice also that when $ltr$ values increases, the freshness results presented by *deferred-immediate* begins to grow (when $ltr = 100$ the freshness is almost doubled compared with the immediate strategies) because there is no simultaneity of tasks as with the immediate strategies. So, when update transaction sizes increase update propagation times rise much more compared to the immediate strategies. This increases freshness values much more seriously.

Response time are impacted more with the *immediate-immediate* strategy (see Figure 9) due to the same reasons already presented in Experiment 1. The difference here is that $\lambda_t << \lambda_q$, increasing the chance of conflict and the response times relative to the *low workloads*. This is true especially when $ltr$ increases. Using MV2PL, query response times for the three strategies in

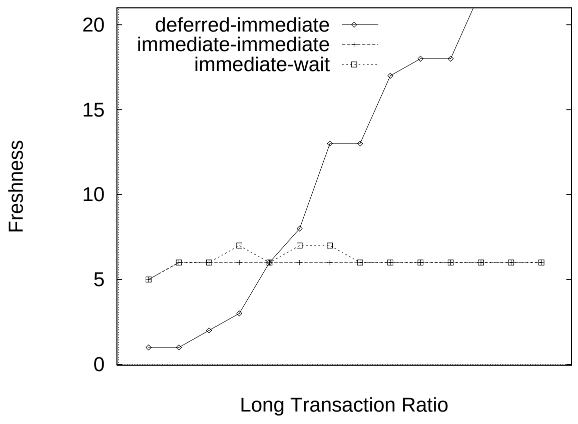

Figure 10: Bursty Workload - Freshness Behavior ($ltr = 100$)

all cases is also reduced to an average of 1.2s with out a significant loss of freshness.

## 5.3 Discussion

With low workloads freshness is only impacted if update transactions are dense and long. In this case *immediate-wait* and *deferred-immediate* show close freshness results because their $t_p$'s are close. *Immediate-immediate* is the strategy that shows the best results because propagation and refreshment are done simultaneously. In conflict situations, query response times depend on $t_{short}$, $\epsilon$ and $\delta$. However, they are not seriously affected because $\lambda_q > \lambda_t$.

In *bursty* workloads, freshness is impacted if update transactions are dense and, in the presence of long transactions freshness is more significantly impacted. The *immediate-immediate* strategy is still the one that presents the best freshness results even with network contention. When all transactions are long, *immediate-wait* performs like *immediate-immediate* because $q_r$ is quickly filled with operations due to the immediate propagation and the effect of the *wait period* of *immediate-wait* may be eliminated. With *immediate-immediate*, query response time may increase in the presence of long transactions and conflicts because $\lambda_q >> \lambda_t$. Query response times are much lower with *immediate-wait* because propagation and refreshment are done in separate tasks. In any case, the use of a multiversion protocol on the slave may significantly reduce query response times, without a sigficant loss of freshness.

In general, *immediate-immediate* is the strategy that presents the best freshness results. However, it may be the one that may suffer the most by network or a master node failure during $RT_i$ propagation, because a data item may be kept locked until the failure is detected. This is the only case

| *Replication Scheme* | *Ownership* | *Propagation* | *Refreshment* |
|---|---|---|---|
| A | Group | Deferred<br>Immediate | Immediate<br>(Reconciliation) |
| B | Master | Deferred<br>Immediate<br>Periodic | Immediate<br>On Demand<br>Group<br>Periodic |

Table 2: Replication Schemes

when $RT_i$ is aborted by the slave.

## 6 Related Work

Table 2 presents two lazy replication schemes and their basic parameters. We use this table to situate our work with respect to others.

Replication scheme A corresponds to a lazy replication scheme where all replica copies are updatable (*update anywhere*). In this case, there is *group ownership* on the replicas. The commom update propagation strategy implemented for this scheme is *deferred-immediate*. A *conflict* happens if two or more nodes update the same replica object. There are several policies for conflict detection and resolution [Gol95, Bob96] that can be based on timestamp ordering, node priority and others. The problem with conflict resolution is that during a certain period of time the database may be in an inconsistent state. Conflicts can not be avoided but its detection may happen earlier by using an immediate propagation.

Replication scheme B is the focus of our work. There are several refreshment strategies for this replication scheme. With *on demand*, each time a query is submitted for execution, secondary copies that are read by the query are refreshed by executing all received refresh transactions. Therefore, a delay may be introduced on query response time. When *group* refresh is used, refresh transactions are executed in groups in accordance with the application's freshness requirements. With the *periodic* approach, refreshment is triggered in fixed intervals. At refreshment time, all received refresh transactions are executed. Finaly, with *periodic propagation*, changes performed by update transactions are stored in the master and propagated periodically. Notice that immediate propagation may used with all refreshment strategies.

*Incremental agreement* is a strategy [CMAP95] that has some features in common with our proposed strategies. However, they focus on managing network failures in replicated databases and do not address the problem of improving freshness. Refreshment is performed using the slave log instead of by the local transaction manager a we do.

The stability and convergence of replication schemes A and B are compared in [GHOS96] through an analytical model. They show that scheme A has unstable behavior as the workload scales up and that using scheme B reduces the problem. However, immediate propagation is not considered. They introduce several concepts that are used in our work and explore the use of mobile and base nodes.

Formal concepts for specifying coherency conditions for replication scheme B in a large scale systems are introduced in [GN95], focusing on the *deferred_immediate strategy*. These concepts permit the calculation of an independent measure of relaxation, called *coherency index*. In this context, there concept of *versions* are closely related to our notion of freshness.

Freshness measures are closely related to *coherency conditions* that are widely explored in [ABGM88, AA95, BGM90] and used in information retrieval systems to define when *cached data* must be updated with respect to changes performed on the central object.

Several derived data refresh strategies are proposed in [AKGM96]: no batching, on demand, periodic and others for a similar scenario. Replica refreshment and derived data refreshment are done in separate transactions. They address freshness improvement, however they focus on the incoherency between derived data and the secondary copy.

Oracle 7 [Pro94] implements an *event-driven* replication, in which triggers on the master tables make copies of changes to data for replication purposes, storing the required change information in tables called *queues* that are periodically propagated. Sybase 10 replication server [Pro94] replicates transactions, not tables, across nodes in the network. The *Log Transfer Manager* implements log monitoring like our approach. However, they do not implement immediate propagation and there is no multi-queue scheme for refreshment.

To our knowlege, ours is the first proposal for an architecture and the parameters to characterize lazy master replication scheme.

## 7 Conclusions

In this paper, we address the problem of freshness in lazy replication schemes We present a framework and an architecture for a specific lazy master replication scheme. The architecture we propose maintains the autonomy of the underlying relational database system. We propose two strategies to improve freshness using our architecture: *immediate-immediate* and *immediate-wait*. The behavior of these strategies is analyzed through practical experimentation and reveals that *immediate_immediate* strategy always improves freshness when compared with *deferred_immediate* and *immediate_wait*. The impacts are most significant for a special type of workload that we call *bursty*, especially when the majority of the update transactions are long. On the other hand, *immediate_wait* only shows results close to the ones revealed by *immediate_immediate* for *bursty* workload in the case where the majority of transactions are long. Using *immediate_wait* in these scenarios avoids aborting transactions in the case of network and site failures. Refresh transaction and view maintenance optimization is possible, permitting even more to improve data freshness.

The downside of using *immediate_immediate* is the increase of query response time due to network delays. However, query response time may be reduced by using *immediate_wait*. Furthermore, it can be drastically reduced by using a multiversion protocol without a signicant loose of freshness. Finally, our analysis reveals that network traffic has significant impact on *immediate_wait*, especially if the majority of the transactions are small.

## Acknowledgments

We would like to thank Dennis Shasha for giving us the motivation of this work, Anthony Tomasic, Francoise Fabret for their precise comments on earlier drafts of this work and Tamer Ozsu for his final review.

## References

[AA95] G. Alonso and A. Abbadi. Partitioned data objects in distriduted databases. *Distributed and Parallel Databases*, (3):5–35, 1995.

[ABGM88] Rafael Alonso, Daniel Barbara, and Hector Garcia-Molina. Quasi-copies: Efficient data sharing from information retrieval systems. *Proceedings of Advances in Data Base Technology (EDBT)*, pages 373–387, 1988.

[AKGM96] Brad Adelberg, Ben Kao, and Hector Garcia-Molina. Database support for efficiently maintaining derived data. *Advances in Database Technology - EDBT*, pages 223 – 240, 1996.

[BGM90] Daniel Barbara and Hector Garcia-Molina. The case controlled inconsistency in replicated data. *Proceedings of the Workshop on Management of Replicated Data*, pages 35 –38, 1990.

[Bob96] Steven Bobrowski. Oracle 7 server concepts, release 7.3. *Oracle Corporation, Redwood City, CA*, 1996.

[CFLS91] M.J. Carey, M.J. Franklin, M. Livny, and E.J. Shekita. Data caching tradeoffs in client-server DBMS architectures. *Proceedings of the ACM SIGMOD*, pages 357–366, June 1991.

[CMAP95] S. Ceri, M.A.W.Houstma, A.M.Keller, and P.Samarati. Independent updates and incremental agreement in replicated databases. *Distributed and Parallel Databases*, 3(3):225–246, July 1995.

[FMS97] Fracoise Fabret, Maja Matulovic, and Eric Simon. State of the art: Data warehouse refreshment. *DWQ - Report*, 1997.

[GHK$^+$97] T. Griffin, R. Hull, B. Kumar, D. Lieuwen, and G. Zhou. A framework for using redundant data to optimize read-intensive database applications. *Proceedings of the International Workshop on Real-Time Databases*, 1997.

[GHOS96] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The danger of replication and a solution. *Proceedings of the ACM SIGMOD*, pages 173–182, June 1996.

[GMS93] H. Gupta, I.S. Mumick, and V.S. Subrahmanian. Maintaining views incremently. *Proceedings of the ACM SIGMOD*, 1993.

[GN95] Rainer Gallersdorfer and Matthias Nicola. Improving performance in replicated databases through relaxed coherency. In *Proceedings of the 21st VLDB Conference*, pages 445 – 456, 1995.

[Gol95] Rob Goldring. Things every update replication customer should know. *Proceeding of the ACM SIGMOD*, (439-440), June 1995.

[GR93] J.N. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan-Kaufmann, August 1993.

[Gra91] Jim Gray. *The Benchmark Handbook*. Morgan Kaufmann, 1991.

[KR87] Bo Kahler and Oddvar Risnes. Extending logging for database snapshot refresh. *Proceedings of the 13th VLDB Conference, Brighton*, pages 389–398, 1987.

[Lad90] Rivka Ladin. Lazy replication: Exploiting the semantics of distributed services. *Proceedings of the Workshop on Management of Replicated Data*, pages 31–34, 1990.

[Moi96] Alex Moissis. *Sybase Replication Server: A Pratical Architecture for Distributing and Sharing Corporate Information*, 1996.

[PHN87] P.A.Bernstein, V. Hadzilacos, and N.Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, 1987.

[Pro94] EDS-Institut Prometheus. Replication et repartition. 1994.

[Sha97] Dennis Shasha. Lessons from wall street: case studies in configuration, tuning and distribution. *Proceedings of the ACM SIGMOD*, pages 498 –501, May 1997.

[SKS86] Sunil K. Sarin, Charles W. Kaufman, and Janet E. Somers. Using history information to process delayed database updates. *Proceedings of the 12th International Conference on VLDB*, pages 71–78, 1986.