ETH zürich Research Collection

Report

Fine-Grained Lazy Replication with Strict Freshness and Correctness Guarantees

Author(s):
Akal, Fuat; Türker, Can; Schek, Hans-Jörg; Grabs, Torsten; Breitbart, Yuri

Publication Date:
2004-09

Permanent Link:
https://doi.org/10.3929/ethz-a-006774941 →

Rights / License:
In Copyright - Non-Commercial Use Permitted →

This page was generated automatically upon download from the ETH Zurich Research Collection. For more information please consult the Terms of use.

ETH Library

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Department of Computer Science
Institute of Information Systems

Fine-Grained Lazy Replication with Strict Freshness and Correctness Guarantees

Fuat Akal
Can Türker
Hans-Jörg Schek
Torsten Grabs
Yuri Breitbart

Technical Report #457, 2004

September 2004

ETH Zurich
Department of Computer Science
Institute of Information Systems

Address:

Fuat Akal, Can Türker, Hans-Jörg Schek
ETH Zurich, Institute of Information Systems
ETH Zentrum, CH-8092 Zurich, Switzerland
{akal, tuerker, schek}@inf.ethz.ch

Torsten Grabs
Microsoft Corp.
One Microsoft Way, Redmond, WA 98052, USA
grabs@acm.org

Yuri Breitbart
Department of Computer Science
Kent State University
Kent, OH 44240, USA
yuri@cs.kent.edu

This report is available via http://www.inf.ethz.ch/publications/ or
ftp://ftp.inf.ethz.ch/pub/publications/tech-reports/

© 2004 Department of Computer Science, ETH Zurich

Fine-Grained Lazy Replication with Strict Freshness and Correctness Guarantees

Fuat Akal, Can Türker, Hans-Jörg Schek
ETH Zurich, Institute of Information Systems
CH-8092 Zurich, Switzerland
{akal,tuerker,schek}@inf.ethz.ch

Torsten Grabs
Microsoft Corp.
One Microsoft Way, Redmond, WA 98052, USA
grabs@acm.org

Yuri Breitbart
Department of Computer Science
Kent State University
Kent, OH 44240, USA
yuri@cs.kent.edu

Abstract

Eager replication management is known to generate unacceptable performance as soon as the update rate or the number of replicas increases. Lazy *replication protocols tackle this problem by decoupling transaction execution from the propagation of new values to replica sites while guaranteeing a correct and more efficient transaction processing and replica maintenance. However, they impose several restrictions on transaction models that are often not valid in practical database settings, e.g., they require that each transaction executes at its initiation site and/or are restricted to full replication schemes. Also, the protocols cannot guarantee that the transactions will always see the freshest available replicas. This paper presents a new lazy replication protocol called* PDBREP *that is free of these restrictions while ensuring one copy serializable executions. The protocol exploits the distinction between read-only and general transactions and works with arbitrary physical data organizations such as partitioning and striping as well as different replica granularities. It does not require that each read-only transaction executes entirely at its initiation site. Hence, each read-only site need not contain a* fully *replicated database.* PDBREP *furthermore generalizes the notion of* freshness *to finer data granules than entire databases. Beside its architectural advantages, experiments revealed that* PDBREP *outperforms related lazy replication techniques.*

1 Introduction

Replication is an essential technique to improve performance of frequent read operations when updates are rare. Updates or any other write operation are challenging in this context since all copies of a replicated data item must be kept up-to-date and consistent, which usually implies additional overhead. Different approaches to replication management have been studied so far. One approach from standard database technology is *eager* replication which uses two-phase commit in combination with two-phase locking to guarantee one-copy serializability and replica coherency [3, 8]. However, Gray et al. [7] have already argued that this approach provides unacceptable performance as soon as the update rate or the number of copies increases. For instance, the deadlock probability is proportional to the fourth power of the transaction size in replicated settings, as shown in [7]. The main drawback with eager replication management is that *all* copies of a data item are maintained within the same database transaction which hinders scalability and prevents from efficient executions. Therefore, eager replication is not a viable approach in most of current data processing environments.

Recent work on *lazy* replication management decouples replica maintenance from the "original" database transaction [6, 5, 10]. In other words, transactions keeping replica up-to-date and consistent run as separate and independent database transactions *after* the "original" transaction has committed. Compared to eager approaches, clearly additional efforts are necessary to guarantee serializable executions. [5, 4] suggest a suite of lazy replication protocols that guarantee one-copy serializable executions. These approaches have lead to significant performance improvements compared to eager replication since multi-site commitment protocols and blocking are avoided. Hence, we will focus on lazy replication in this paper.

Previous work on lazy replication has concentrated on performance and correctness only. In particular, it did not consider that important practical scenarios may require up-to-date data – a property that is not necessarily satisfied with lazy replication techniques. Recent work by Röhm et

al. [14] addresses this issue with the additional notion of *freshness*. It allows read-only clients to define *freshness requirements* stating how up-to-date data items must be when being accessed. However, the approach of [14] suffers from several important shortcomings. First, it requires a centralized coordination component for scheduling and bookkeeping which is a potential bottleneck and a single point of failure. Second, [14] has only considered full replication at a granularity of complete databases. Clearly, this is way too inflexible as it precludes more sophisticated physical data organization schemes such as partial replication, partitioning or striping across sites. Third, previous work on lazy replication like [11, 15, 9, 4] assumed that the transaction executes entirely at its initiation site, which may not be the case in practical database settings.

The objective of our current work is to attack the aforementioned shortcomings and to build a system that covers the following requirements jointly: (1) performing replica maintenance by a lazy approach, (2) ensuring correct executions, (3) allowing users to specify freshness requirements, (4) supporting arbitrary physical data organization schemes, and (5) executing read-only transactions at several data sites in parallel.

Clearly, to devise an approach and to build a respective system which covers all of the above requirements jointly and efficiently would constitute an important step in replication management. However, it is by far not obvious how to generalize previous work in this respect. The following example illustrates that lazy replication as proposed in [4] fails already if one allows a transaction to read data items from several sites.

Example 1: We assume that for each data item there is a single primary site responsible for updates to the data item. We call such site primary. All other copies of the data item are called secondary. Let us assume that there are four sites s_1, s_2, s_3, and s_4, which are interconnected by a communication network, as shown in Figure 1.

s_1 and s_2 contain data items a and b with s_1 being a primary for a and s_2 being primary for b. s_3 and s_4 in turn store secondary copies of a and b, each. Further suppose that at s_1 (s_2) a transaction T_1 (T_2): $w_1(a)$ ($w_2(b)$) is submitted. At about the same time at sites s_3 and s_4, *read-only* transactions T_3 and T_4 are submitted, where

T_3: $r_3(a) r_3(b)$
T_4: $r_4(a) r_4(b)$

Suppose $r_3(a)$ and $r_4(b)$ are scheduled to be executed at s_3, and $r_3(b)$ and $r_4(a)$ at s_4. In this case, the following local schedules can be generated at sites s_3 and s_4, respectively:

S_3: $w_1(a)\ r_3(a)\ w_2(b)\ r_4(b)$
S_4: $r_4(a)\ w_1(a)\ r_3(b)\ w_2(b)$

where $T_1' = T_1''$: $w_1(a)$ and $T_2' = T_2''$: $w_2(b)$ are *propagation* transactions generated by T_1 and T_2 at s_3 and s_4, respectively. Figure 1 illustrates these read-only and propagation transactions and their conflicts. Clearly, the global

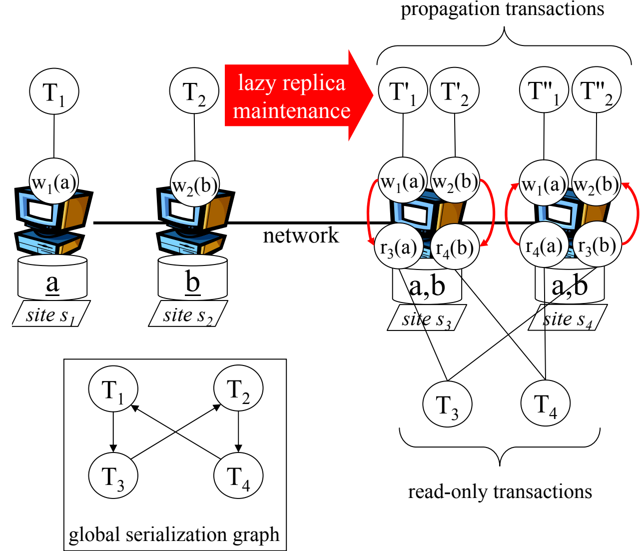

Figure 1. Lazy replication management

schedule we have obtained is not globally serializable since the global serialization graph is cyclic. Observe, however, that at each site propagation transactions are executed in the same order [14] and, furthermore, each site-specific schedule is locally correct. ◇

As our previous discussion and the example above show, approaches from state-of-the art database systems or previous research are not viable in order to cover all aforementioned requirements jointly. The objective of our work is to keep the overhead of replica maintenance as small as possible and at the same time to guarantee good performance of read-only transactions with a requested level of data freshness. To achieve these goals, we developed a new protocol for lazy replication management, called PDBREP[1]. The main idea of PDBREP is to exploit unique version identifiers to guarantee efficient replica maintenance with correct schedules and up-to-date results. Moreover, PDBREP exploits two important characteristics, namely (1) the distinction between *read-only transactions* versus *update transactions*, and (2) a partitioning of the sites into *read-only sites* versus *update sites* to process read-only and update transactions, respectively. Globally correct execution of update transactions over update sites is already covered in previous work, e.g. [4]. It is therefore not the concern of this paper. Instead, the objective of our work is to schedule read-only transactions and replica maintenance at the read-only sites in a globally correct way with fault tolerance guarantees.

Our main contributions are as follows:

- PDBREP supports arbitrary physical data organization schemes ranging from full replication at the granular-

[1] PDBREP stands for the replication protocol we implemented within the PowerDB project at ETH Zurich [16]

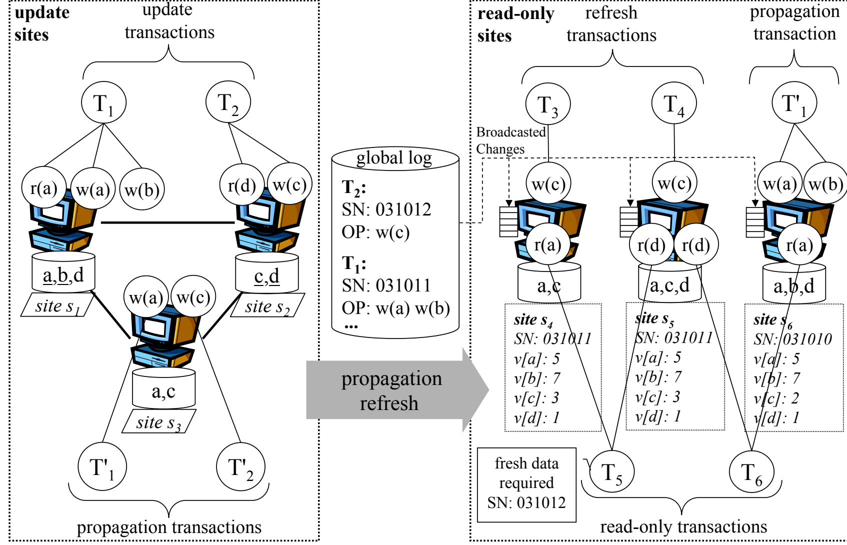

Figure 2. System architecture underlying PDBREP

ity of complete databases to partial replication combined with partitioning and striping.

- PDBREP respects user-specified freshness requirements. It guarantees that queries only process data that is at least as up-to-date as specified by the freshness degree. This includes the special case where users always work with up-to-date data.
- We show that PDBREP produces correct, i.e., one-copy serializable, global executions.
- PDBREP does not require a centralized component for coordination.
- Our experiments reveal that PDBREP outperforms related lazy replication protocols.

The remainder of this paper is organized as follows: Section 2 presents the underlying system model, describing the kinds of transactions and database cluster nodes. Section 3 introduces PDBREP in detail, while Section 4 presents performance results. Section 5 concludes the paper.

2 System Model

We consider a replicated database that contains data items a, b, ... which are distributed and replicated among the sites s_1, $s_2, \ldots, s_t$.[2] For each data item a there is a *primary site* denoted by $p(a)$. If site s contains a replica of a and $s \neq p(a)$, we call a replica of a at s a *secondary copy*. For instance, site s_1 holds a primary (underlined) copy of a while site s_3 only stores a secondary (non-underlined) copy of a.

[2] Since we use relational database systems, data items are (partitions of) relations while operations are queries or data manipulation statements.

Following previous work [5, 4], updates of a data item first occur at its primary site, and only after that these updates are propagated to each secondary site. A simple consequence of this fact is that all write operations to the same data item can be ordered according to the order of their execution at the primary site. Similarly to [14], we partition all sites into two classes: (1) read-only, and (2) update. At *read-only sites*, we run exclusively read-only transactions. Consequently, it follows that read-only sites do not contain any primary copies. At *update sites* any transaction can run. Figure 2 illustrates the distinction between site types.

2.1 Transaction Types

There are four types of transactions in our model: *update*, *read-only*, *propagation*, and *refresh* transactions. Based on the protocols discussed in [4], an *update transaction* T may update data item a if T is initiated at the primary site of a. T, however, may read any data item at this site.

Example 2: Figure 2 shows two update transactions T_1 and T_2. As the figure shows, update transactions only run on update sites and write operations of update transactions only occur at primary sites. $\diamond$

Read-only transactions in turn may be initiated at any read-only site and may read data items from any read-only site. Besides, separate read operations of a read-only transaction may run at several different read-only sites. This is an important generalization of [14] which has only considered full replication and local read-only transactions. Moreover, it allows for arbitrary physical data organizations at the read-only sites and routing of read-only operations [2, 13].

Example 3: Consider again the architecture shown in Figure 2. It shows two read-only transactions T_5 and T_6. As the

figure shows, read-only transactions only run on read-only sites, and they may distribute their operations over several sites – depending on the physical data organization or the workload situation at read-only sites. ◇

Propagation transactions are the third transaction type in our model. These transactions continuously propagate changes occurring at the primary sites to the secondary copies.[3] By virtue of our model, the propagation transactions for the same data item are initiated from the same primary site. As a result, all updates at secondary sites of the same data items are ordered by the order of the primary transaction that performed such an update at the primary site of the data item.

Example 4: Our running example in Figure 2 shows the propagation transaction T_1' propagating the changes of update transaction T_1 to site s_6. It comprises all write operations of T_1, but not its read operations. ◇

Finally, there are *refresh transactions* that bring the secondary copies at read-only sites to the up-to-date state as compared to the primary copies. A refresh transaction aggregates one or several propagation transactions into a single bulk transaction. Thus, it only comprises write operations. A refresh transaction is processed when a read-only transaction requests a version that is younger than the version actually stored at the read-only site. Thus, a refresh transaction is generally requested by the site's transaction manager, when a freshness requirement of the read transaction at the site cannot be satisfied. The propagation transaction, on the other hand, is not requested by the site's transaction manager. This is the major difference between propagation and refresh transaction types.

Example 5: Consider again our running example. It shows two refresh transactions T_3 and T_4. They bring sites s_4 and s_5 to an up-to-date state. This is needed to run the read-only transaction T_5 that requires fresh data with at least a timestamp of 031012. The timestamps of sites 4 and 5 are equal to 031011. The system therefore checks the propagation queues for potentially missing updates – in the figure this is the update transaction T_2 with timestamp 031012. The system runs the refresh transactions T_3 and T_4 on sites 4 and 5 in order to incorporate the still missing write operations to these sites. After completing the refresh, the read-only transaction T_5 is executed at the sites. ◇

Update, propagation, and refresh transactions execute as decoupled independent database transactions. We assume that each site ensures locally correct (serializable) transaction executions. To guarantee global correctness, we adopt the criterion *one-copy-serializability* [3] such that, regardless of the number of read-only sites, read-only transactions always see a consistent database state as if there were a single database only. We will come back to this issue later in Section 3.3.

[3] Note that propagation transactions can also be "bulked".

3 The PDBREP Protocol

3.1 Overview of the Protocol

PDBREP keeps secondary copies at the read-only sites up-to-date by continuously scheduling propagation transactions as update transactions at the update sites commit. We assume that a globally serializable schedule is produced for all update sites by some algorithm that is of no concern for this paper. Furthermore, the update transactions serialization order is their commit order. Thus, each propagation transaction inherits the number in which update transaction committed in the global schedule and this number we call the propagation transaction sequence number. Consequently, each propagation transaction has a unique sequence number that is known to each read-only site.

To ensure correct executions at read-only sites, each read-only transaction determines a version of the data items it reads at its start. PDBREP foresees two different approaches to determine this version. The first so-called *implicit* approach determines the versions of the data items accessed by a read-only transaction T from the versions of the data items at the sites accessed by T. With the second so-called *explicit* approach, users define the *freshness* of the data accessed by their read-only transactions as a quality of service parameter when submitting their transactions. If a read-only transaction is scheduled to a site that does not yet fulfill the freshness requirements, a refresh transaction updates the data items at that site similar to [14].

With either of the two aforementioned approaches, freshness locks, which behave similar to ordered shared locks [1], are placed on the data items at the read-only sites ensure that ongoing replica maintenance transactions do not overwrite versions which are still needed by ongoing read-only transactions. With PDBREP, transactions always operate on transactionally consistent versions. As we prove in Subsection 3.3.2, PDBREP ensures one-copy serializable executions of propagation, refresh, and read-only transactions.

3.2 Definition of the Protocol

After having informally sketched PDBREP, we are now ready to formally state the protocol. First, we introduce the following definitions.

Definition 1 (Sequence Number) *Update transactions can be ordered sequentially by their commit order on the update sites. For an update transaction T, we define its* sequence number *$SN(T)$ as its commit timestamp. Let P now denote a propagation transaction for T, then the sequence number $SN(P)$ is defined as $SN(P) = SN(T)$.*

Each propagation transaction at each site then has a unique sequence number. However, as example 1 shows, executing propagation transaction in the order of their sequence numbers does not guarantee by itself one-copy serializability.

Definition 2 (Update Counter Vector) *Let n be the number of different database items. The global log site s_{gl} and each read-only site s maintains an* update counter vector *v of dimension n defined as follows. For any data item a stored at some site in the system, $v(s)[a]$ stores the number of committed propagation transactions that have updated a at s and, $v(s_{gl})[a]$ stores the number of committed updates that have been performed on primary copy of a.*

Note that by this definition any vector $v(s)$ has a component for a even though a is not necessarily stored on s. Moreover, vectors v may differ between sites while propagation transactions are ongoing. Figure 2 represents these vectors together with the site number. Initially, all vector components at all sites are zero.

Definition 3 (Version Vector of a Transaction) *We define the* version vector of transaction *T as $v(T)$ similar to the update counter vector of sites, i.e., $v(T)$ has a counter $v(T)[a]$ for each data item a that occurs in the overall system.*

We include an additional quality-of-service parameter to allow read-only transactions to specify what freshness of data they require.

Definition 4 (Freshness of Data) Freshness *is defined by means of the timestamps at the propagation queue. Clients can refer to these timestamps in order to specify the* freshness requirements *of their requests. We say that a site is* fresh *if its timestamp is equal to or younger than the timestamp specified by the client.*

Let TS_T, TS_s, and S_T denote the freshness requirement of transaction T, the current timestamp of site s, and the set of sites involved in the execution read-only transaction T, respectively.

Definition 5 (Required Freshness Level) *If $TS_T > TS_s$ holds, then the* refresh transaction *R brings all $s \in S_T$ to the same freshness level fl by executing a sequence of write operations. It also includes maintenance of the site update counter vectors. Therefore, running a refresh transaction has the same effect as sequentially running the remaining propagation transactions at the site. With the implicit approach, i.e. when the user has not specified a freshness requirement, fl is given by the freshness level of the freshest site in S_T. With the explicit approach, fl is determined by the maximum of the user-specified freshness level and the level as determined in the implicit approach. If $TS_T \leq TS_s$ holds, then the refresh transaction R is empty.*

A refresh transaction as defined above is executed at every site s accessed by a read-only transaction T if $TS_T > TS_s$ holds.

We introduce the concept of freshness locks to hinder propagation and refresh transactions that bring a data item to a freshness level that is above of the freshness level of a read transaction.

Definition 6 (Freshness Lock) *A* freshness lock *is placed on a data item a at site s with an associated freshness timestamp ft demanded by the acquiring read transaction. Such a lock disallows a write operation on data item a on site s if this operation brings data item a on site s to a higher freshness timestamp than ft.*

Definition 7 (Compatibility of Locks) *Freshness locks are compatible, i.e., freshness locks on the same data item can be acquired by different read transactions possibly with different freshness timestamps.*

Altogether, PDBREP comprises the following rules to schedule transactions on read-only sites:

1. Propagation transactions execute changes from the local propagation queues in the order of the sequence numbers. A site s delays a propagation transaction P until all propagation transactions with smaller sequence numbers than P have committed at s. Missing changes are detected by non-continuous sequence numbers, and the missing propagation transactions are run with the changes fetched.

2. A propagation transaction is dropped if the site has already seen and processed a propagation transaction with the same or higher sequence number.

3. Each read-only transaction submitted at site s requests freshness locks with its required (freshness) timestamp at s for all data items read by the transaction. If a data item does not exist at s, then the freshness lock is requested from a site that has a copy of the data item. The transaction waits for the freshness locks to be granted.

   Note that we do *not* require *all* copies of a data item to be freshness locked, but *only one*. Furthermore, note that several freshness locks with different (freshness) timestamps can be placed on the same data item.

4. If transaction T_j requires freshness with a timestamp $TS(T_j)$ and none of the sites locked for T_j meets the freshness requirement of T_j, then all sites involved in T_j are refreshed according to Definition 5. [4]

5. After freshness locking and refresh, a transaction T_j receives the version vector $v(T_j) = max\,(v(s_i) \mid i \in sites(T_j))$ where $sites(T_j)$ are the sites T_j has placed freshness locks on. [5]

6. A read-only transaction T_j may submit a read operation $r_j(a)$ to any read-only site s_k that stores a copy of a. If $r_j(a)$ is submitted to some such site s_k, then the following rules apply:

[4] If at least one site is fresh enough, then the transaction gets a counter vector that will enforce this freshness for all other sites implicitly.

[5] We will prove later that all version vectors generated by PDBREP are actually comparable.

Algorithm 1: Scheduling read-only transactions

```
Data      : read-only transaction T_i = {op_i};
            user-demanded freshness timestamp ft,
            ft = 0 means implicit approach;
// place freshness locks
foreach op ∈ {op_i} do
    let a denote the data item read by op;
    acquire lock on a at some site s_j storing a copy of a;
    S := S ∪ s_j;
end
// compute transaction's version vector and timestamp
if ft > max (ts_{s_j} | s_j ∈ S) then
    // none of the involved sites meets required freshness
    v(T) = v(s_gl);
    // global version vector to enforce freshness 1.0
    ts = timestamp(s_gl);
else
    v(T) = max (v(s_l) | s_l ∈ S);
    // version vector of the freshest site involved in T_i
    ts = timestamp(s_l);
end
// execute the operations
foreach op ∈ {op_i} do
    let s_j denote the site op is routed to;
    let a denote again the data item op reads;
    BOT;
    switch v(s_j) do
        case v(s_j)[a] = v(T)[a]
            process op at s_j;
        case v(s_j)[a] < v(T)[a]
            // refresh site s_j according to the required
            // timestamp ts and maintain local counters
            refresh s_j to ts;
            process op at s_j;
        case v(s_j)[a] > v(T)[a]
            // let s_k the site with the preclaimed lock
            process op at s_k;
    end
    EOT;
end
// commit processing
foreach s_i ∈ S do
    release T's freshness locks at site s_i;
end
```

(a) If $v(s_k)[a] = v(T_j)[a]$, the read operation is executed.

(b) If $v(s_k)[a] < v(T_j)[a]$, then s_k runs a refresh until it reaches $v(s_k)[a] = v(T_j)[a]$. Then, the read operation can be processed on s_k.

(c) If $v(s_k)[a] > v(T_j)[a]$, then the read operation is re-routed to another site s_l with $v(s_l)[a] \leq v(T_j)[a]$. Note that at least the site with the preclaimed lock of T_j on a satisfies the latter condition.

Algorithm 2: Scheduling propagation transactions

```
Data      : propagation transaction P, its operations
            {op_i}, site s_i
BOT;
// check site sequence number
if SN(P) ≤ SN(s_i) then return;
// execute the operations
foreach op ∈ {op_i} do
    let a denote the data item op writes;
    if a is locked ∧ v(s_j)[a] ≥ min (v(T)[a] | T ∈ T_a(s_j))
    then wait for lock release;
    process op;
end
// commit processing
foreach data item a accessed by P do
    v(s_i)[a] := v(s_i)[a] + 1;
end
SN(s_i) := SN(P);
EOT;
```

7. If $w_n(a)$ is submitted at s_k, then the following rules apply:

   (a) If there is no freshness lock on the data item a, then the operation is executed.

   (b) If $v(s_k)[a] < min\,(v(T)[a] \mid T \in \mathcal{T}_a(s_k))$ where $\mathcal{T}_a(s_k)$ denotes the set of all read-only transactions holding a freshness lock on the data item a on s_k, then the operation is executed.

   (c) Otherwise, the operation is delayed until the conflicting freshness locks are released at the site, i.e., one of the above two holds true.

8. If a commit is submitted, it is executed. With a read-only transaction, this also releases all of its freshness locks. The commit of a propagation transaction P at site s also maintains the update counters in $v(s)$, the sequence number, and the timestamp at s as follows:

   - If P has updated data item a, $v(s)[a]$ is incremented by one.
   - If P comprises a write operation on data item a, but data item a is not stored at s, $v(s)[a]$ is incremented by one.
   - Otherwise, $v(s)[a]$ remains unchanged.
   - SN_s is overwritten with the value of SN_P.

Note that propagation transactions – according to the lazy replication scheme – run locally as independent database transactions at each site. With read-only transactions in turn, an even more fine-grained database transaction granularity is feasible. This helps to avoid low-level lock contention at the database systems. Algorithms 1 and 2 summarize the algorithms for scheduling read-only and propagation transactions and indicate local database transaction boundaries by BOT and EOT, respectively.

3.3 Serializability

The objective of this section is to prove that the PDBREP protocol as defined above guarantees one-copy serializable schedules for propagation and read-only transactions. In Subsection 3.3.1, we start by introducing several notions and definitions required for the proof. Subsection 3.3.2 in turn conducts the proof of serializability.

3.3.1 Preliminaries and Definitions

Definition 8 *In analogy to the version read by a read-only transaction, we define the* version vector $v(P)$ of a propagation transaction P *executed at a read-only site* s *as the vector of the update counter values at* s *immediately after* P *has committed.*

This allows us to compare the versions of any pair of read-only or propagation transactions.

Definition 9 *The* comparison *of two* versions v_i *and* v_j *is defined as follows:*

$$v_i = v_j \quad := \quad \forall k : v_i[k] = v_j[k] \tag{1}$$
$$v_i < v_j \quad := \quad \neg(v_i = v_j) \wedge \forall k : v_i[k] \leq v_j[k] \tag{2}$$
$$v_i > v_j \quad := \quad \neg(v_i = v_j) \wedge \forall k : v_i[k] \geq v_j[k] \tag{3}$$

If one of the three above is true we say that two versions are comparable.

It will be one of the important steps in our subsequent proof that PDBREP only produces comparable versions. Note that the above definition also allows for non-comparable versions where neither of the above comparisons yields true.

Example 6: An example of two non-comparable versions v_1 and v_2 is as follows:

$$v_1 = \begin{pmatrix} 1 \\ 2 \end{pmatrix} \qquad v_2 = \begin{pmatrix} 2 \\ 1 \end{pmatrix}$$

◇

3.3.2 Proof of Serializability

For executions of read-only and propagation transactions at read-only sites to be correct, the global schedule must be serializable. Note that we did assume that local schedules are correct. Therefore, we do not care about local serializability. It is granted by the DBMS used. Note further that our focus is on executions at the read-only sites, i.e., we do not consider executions at the update sites. We conduct the proof of global serializability by contradiction in several steps:

1. Our initial assumption is that the PDBREP protocol does not produce one-copy serializable schedules.
2. We then show that read-only transactions in isolation as well as propagation transactions in isolation cannot lead to non-serializable global schedules. Therefore, non-serializable schedules must comprise both read-only and propagation transactions.
3. All versions generated by PDBREP are comparable and PDBREP establishes a total order across the versions.
4. An edge between two transactions in the global serialization graph maps to a comparison of versions.
5. The last steps then is to show that mapping any non-serializable schedule to a sequence of version comparisons leads to a contradiction.

Step 1. Let S be a global schedule resulting from the execution of read-only and propagation transactions at the read-only sites. Let us assume that the PDBREP protocol allows for executions that are not one-copy serializable. Then there is a cycle in the global serialization graph $SG(S)$ of schedule S [12]. Let

$$C := T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow \ldots \rightarrow T_n \rightarrow T_1$$

denote the cycle in $SG(S)$.

Step 2. Each transaction in S consists of either only read or only write operations. We now show that only a mix of read-only and propagation transactions can yield a cyclic serialization graph. We first state that read-only transactions in isolation cannot produce non-serializable schedules.

Lemma 1 (Read-Only Serializability) *Each global schedule comprising only of read-only transactions is serializable.*

Proof. By definition, conflicts always involve write operations. Since a read-only transaction does not comprise write operations there are no edges in the global serialization graph. Consequently, $SG(S)$ of a global read-only schedule S is acyclic. □

Moreover, propagation transactions in isolation cannot lead to inconsistent executions, either. We state this observation with the following lemma.

Lemma 2 (Propagation Serializability) *Each global schedule comprising only of propagation transactions is serializable.*

Proof. Since propagation obeys the commit order as established by the sequence numbers, there cannot be a cycle in the global serialization graph $SG(S)$. □

Since propagation alone does not lead to non-serializable global schedules, at least one transaction in C must be a read-only transaction. Without loss of generality, let T_1 denote such a read-only transaction in C.

Step 3. The following lemma states two important properties for any pair of versions generated by PDBREP:

Lemma 3 (Total Order of Versions) *Any two versions v_i and v_j at any of the read-only sites are comparable according to Definition 9. Moreover,* PDBREP *establishes a total order of versions.*

Proof. The serialization order of propagation transactions is established by the sequence numbers and then enforced at all other read-only sites (Rule 1). This prevents from situations where propagation transactions P_1 and P_2 are serialized in different orders at different sites which would lead to an incomplete ordering of versions. □

Step 4. Another important observation for PDBREP is the effect of conflicts onto the order of versions, as stated in the following lemma.

Lemma 4 (Version Sequence (VS)) *If an edge $T_i \rightarrow T_j$ exists in the global serialization graph $SG(S)$, then also $v(T_i) \leq v(T_j)$.*

Proof. An edge between T_i and T_j occurs in $SG(S)$ if and only if some operation of T_i is in conflict with an operation of T_j ($i \neq j$). We distinguish the following cases:

- read-write (RW) conflict: T_i reads a data item that is overwritten by T_j. Now, recall the properties of total version ordering and that read-only transactions read from a single well-defined version. With these two properties, it follows that $v(T_i) < v(T_j)$.
- write-read (WR) conflict: T_i writes a version of a data item that is read by T_j. Rule 6 guarantees that read-only transactions read only the same version of data items, namely $v(T_j)$. Together with the property of total version ordering, it follows that $v(T_i) \leq v(T_j)$.
- write-write (WW) conflict: T_i writes a version of a data item that is overwritten by T_j. Together with the property of total version ordering it follows that $v(T_i) < v(T_j)$.

Summarizing the three above cases yields:

$$T_i \rightarrow T_j \in SG(S) \Rightarrow v(T_i) \leq v(T_j)$$

which proves the proposition of our lemma. □

Step 5. We are now ready to state the serializability of schedules produced by PDBREP in the following theorem:

Theorem 1 (Serializability of PDBREP schedules) *The* PDBREP *protocol produces serializable schedules for read-only and propagation transaction on read-only sites, i.e., the global serialization graph of those schedules is acyclic.*

Proof. Applying the version sequence property defined in Lemma 4 to all edges in the cycle C leads to the following expression:

$$VS(C) := v(T_1) \leq v(T_2) \leq v(T_3) \leq \ldots \leq v(T_n) \leq v(T_1)$$

Let us now investigate the read-only transaction T_1 in more detail. Clearly, T_1 can only conflict with propagation transactions since any interleaving of T_1 with other read-only transaction is conflict-free and only read-only transactions and propagation transactions are allowed to execute at read-only sites. Therefore, only RW-edges (RW: read-write conflict) and WR-edges (WR: write-read conflict) can connect T_1 to the other transactions T_n and T_2 in the cycle C. Since T_1 has an incoming and an outgoing edge in the cycle, we distinguish the following cases:

- $T_1 \rightarrow T_2$ (outgoing edge): Since T_1 can only conflict with propagation transactions, T_2 must be a propagation transaction. Hence, this edge represents a RW-conflict and T_2 overwrites a data item that T_1 has read. Therefore, $v(T_1) < v(T_2)$ where $v(T_2)$ is the version generated by executing T_2.
- $T_n \rightarrow T_1$ (incoming edge): For the same argument as above, T_n must be a propagation transaction. Hence, this edge represents a WR-conflict and T_1 reads a data item that T_n has written, i.e., T_1 reads from T_n. Therefore, $v(T_n) \leq v(T_1)$ where $v(T_n)$ is the version generated by executing T_n.

Substituting the version property for the outgoing edge of T_1, i.e., $v(T_1) < v(T_2)$, into $VS(C)$ leads us to the following expression:

$$VS(C)' := v(T_1) < v(T_2) \leq v(T_3) \leq \ldots \leq v(T_n) \leq v(T_1)$$

It yields $v(T_1) < v(T_1)$ which is a contradiction to the consistent version that a read-only transaction must read according to Rule 3 of the protocol. This invalidates our initial assumption of a cyclic serialization graph. Moreover, it proves that global serialization graphs generated by the PDBREP protocol are acyclic and executions of read-only and propagation transaction on read-only sites are one-copy serializable. □

3.4 Discussion

The above presentation has shown that PDBREP keeps copies of replicated data items on read-only sites consistent. Moreover, global schedules of PDBREP for propagation and read-only transactions are one-copy serializable. The following discussion highlights further important characteristics of PDBREP. In particular, we discuss that PDBREP is deadlock-free, but can lead to waiting for read-only transactions. We also discuss how PDBREP avoids aborts of read-only transactions, and its resilience to site failures.

Global Deadlocks. With PDBREP, global deadlocks, i.e., deadlocks at the level of PDBREP scheduling, cannot occur. This is because database transactions are only local and because PDBREP locks cannot lead to cyclic waiting conditions. We formulate this as the following lemma:

Lemma 5 *Scheduling with* PDBREP *does not produce global deadlocks.*

Note that we do not need to consider deadlocks at the level of database transactions for two reasons. (1) There are no distributed two-phase-commit database transactions with PDBREP. (2) The database systems at the sites resolve local deadlocks locally so that PDBREP may have to restart some database transactions. For these reasons, only deadlocks at the level of PDBREP scheduling are to be considered. To prove the above lemma correct we further rely on the following definition at the level of PDBREP scheduling:

Definition 10 (Local and Global WFG) *A* local wait-for-graph at the PDBREP level ($LWFG_s(S)$) *is defined for each read-only site* s *and a local schedule* S*. The vertices of* $LWFG_s(S)$ *are the propagation and read-only transactions running at* s*. There is an edge between* T_i *and* T_j *in* $LWFG_s(S)$ *if* T_j *cannot execute due to a* PDBREP *lock until* T_i *has finished. The* global PDBREP wait-for-graph ($GWFG$) *is defined as the union of the local wait-for-graphs over all sites.*

We are now ready to prove Lemma 5.

Proof. We conduct the proof by contradiction over properties of the versions. We start with the assumption that there were a deadlock, i.e., that the global wait-for-graph were cyclic.

$$GWFG := T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n \rightarrow T_1$$

Further analysis of an arbitrary edge $T_i \rightarrow T_j$ in $GWFG$ leads to the following cases:

- T_i (read-only transaction) $\rightarrow T_j$ (propagation transaction): T_i has placed a freshness lock on some site s and T_j cannot proceed because this would overwrite the version T_i needs. This implies that T_i reads from a lower version than the one that T_j produces, i.e., $v(T_i) < v(T_j)$.
- T_i (propagation transaction) $\rightarrow T_j$ (read-only transaction): T_j has placed a freshness lock on some site s and T_j cannot proceed because the appropriate version is not yet available. Hence, the version produced by T_i is lower or at least equal to the version T_j requires, i.e., $v(T_i) \leq v(T_j)$.
- $T_i \rightarrow T_j$ (both propagation transactions): T_j cannot start because T_i has not finished yet and T_i has a lower sequence number than T_j. Consequently, $v(T_i) < v(T_j)$.
- $T_i \rightarrow T_j$ (both read-only transactions): Read-only transactions do not wait for other read-only transactions. Hence, this case cannot occur with PDBREP.

A first consequence of this analysis is that read-only transactions in isolation as expected cannot lead to global deadlocks. A further observation is that propagation in isolation cannot produce global deadlocks either because this would lead to a contradiction over the versions when substituting the version expressions derived into the $GWFG$:

$$v(T_1) < v(T_2) < \ldots < v(T_n) < v(T_1)$$

We will now show that the general case where both propagation and read-only transactions occur in the $GWFG$ leads to a similar contradiction. Let us first assume without loss of generality that T_1 is a read-only transaction. This implies that both T_n and T_2 are propagation transactions since edges between two read-only transactions cannot exist in $GWFG$. When substituting the version relations for read-only and propagation transactions as derived above into the $GWFG$ we get the following expression:

$$v(T_1) < v(T_2) \leq \ldots \leq v(T_n) < v(T_1)$$

This yields $v(T_1) < v(T_1)$ - a contradiction to the consistent version property of read-only transactions. This proves our initial assumption of a cyclic global wait-for-graph wrong and proves the correctness of Lemma 5 that global deadlocks cannot occur with PDBREP. □

Effect of Refresh Transactions. Fortunately, none of the previous statements will be invalidated due to the following considerations: A refresh transaction is always started on behalf of and inside of a read-only transaction. In this way, according to Definition 5, all involved sites will consistently increase their version counters. For the duration of a read-only transaction T no other read-only transaction T' with a higher freshness requirement can be executed since the refresh transaction of T' cannot overwrite versions needed by ongoing transaction T due to freshness locks. Considering now the interleaving between propagation transaction P and read-only transaction T including a refresh transaction inside, we distinguish two cases:

1. If $TS_P \leq TS_{S_T}$, then P is simply skipped.
2. Otherwise, P waits until T finishes.

From this, we conclude that no additional dependencies are introduced into the global serialization graph.

Avoiding Aborts of Read-Only Transactions. With PDBREP, it is not necessary to abort read-only transactions due to overwritten versions. This is because PDBREP requires a read-only transaction T to preclaim locks on the data items accessed by T. Freshness locks ensure that propagation transactions can only overwrite a version if it is no longer needed for an active read-only transaction. Consequently, there is always a node that either still keeps the appropriate version or that waits until the appropriate version becomes available through propagation. Hence, there is no need to abort a read-only transaction due to a missing version under normal circumstances. Site failures, however, may require to abort a transaction if the site with the needed version does not come back.

4 Experimental Evaluation

We have implemented a full-fledged prototype on which we ran a series of experiments to evaluate the performance characteristics of PDBREP.

4.1 Experimental Setup

The prototype comprises a cluster of databases which among others contains a designated OLTP node, a global log, a distributed coordinator layer and a client simulator. The evaluation has been conducted on a cluster of databases consisting of 64 PCs (1 GHz Pentium III, 256 MBytes RAM and two SCSI harddisks) each running Microsoft SQL Server 2000 under Windows 2000 Advanced Server. All nodes are interconnected by a switched 100 MBit Ethernet.

In the experiments, we used the TPC-R benchmark [17]. We generated the database according to the TPC-R benchmark with a scaling factor 1. We compared two PDBREP settings:

- In the first setting, the read-only sites are refreshed only on demand of queries by using bulked refresh transactions, i.e., the continuous propagation of transactions and local input queues of PDBREP is switched off. In this setting, PDBREP exactly corresponds to the CRM (*Coordinated Replication Management*) approach proposed in [14]. In the following, we therefore refer to this PDBREP setting as CRM.
- In the second setting, updates that occur at update sites are continuously broadcasted to read-only sites and enqueued in local queues. In addition to refresh transactions similar to CRM, these enqueued changes are also bulked into propagation transactions and applied to the site when it is possible. In contrast to CRM, refresh transactions exploit localized changes by continuous broadcasting, too. In the following, we will refer to this more general setting simply as PDBREP.

For the bulked operations, we considered the update rate at update sites as the unit size of bulked updates. That is, all updates performed in one second are bulked into a package and that package is applied to the database in a single propagation transaction. We also modeled refresh transactions as a bulked sequence of these packages. The update rate used in the experiments was 280 updates per second.

For the freshness degree, we used different freshness degrees ranging from 1.0 to 0.6 to express how fresh data we demand. We simply mapped these freshness degrees to the timestamps of update transactions, i.e., when a read transaction is initiated the timestamp of the last committed update transaction corresponds to freshness degree 1.0 for that transaction. That is, freshness 1.0 refers to the freshest version of a data item while the lower freshness degrees are computed based on the proportion of the propagated updates with respect to the overall amount of updates at the update site.

In the comparisons, we varied two parameters. The first one is the freshness degree of data requested by a query. The second one is the workload of the cluster nodes to show the utilization of the cluster. For a certain period of time (measurement interval), it basically tells how many percent of the cluster is used. There are three different workloads: 100%, 75%, and 50%. Workload 100% means that cluster is fully busy with evaluating queries all the time, i.e., the cluster nodes are never idle since queries arrive continuously. This workload models the peak time of an OLAP system. On the other hand, workload 50% says that cluster is idle for half of the measurement interval, i.e., queries arrive after some time breaks.

4.2 Outcomes of the Experiments

The first experiment shall reveal how the workload and freshness requirement influence the query performance in case of PDBREP and CRM. For the comparison, we used the three different workloads to evaluate query streams for five different freshness degrees from 0.6 to 1.0. The results are depicted in Figure 3. They show that PDBREP performs better than CRM for all three workloads and five freshness degrees.

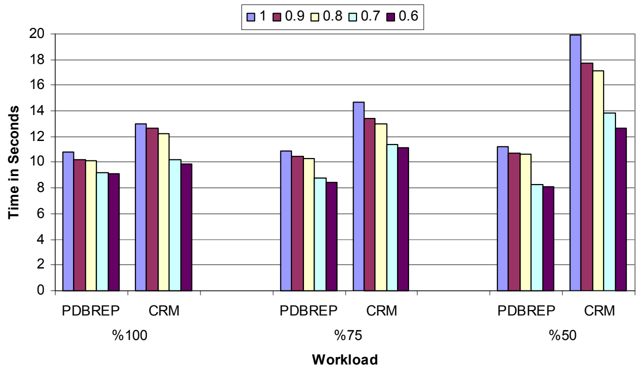

Figure 3. Average Query Evaluation Times for Different Workloads and Freshness Values

When the cluster is 100% loaded, i.e., all nodes are busy with evaluating a query, queries are performed 17% faster in average with PDBREP as compared to CRM. This is primarily due to the fact that PDBREP exploits the advantage of fetching (at least part of) changes from the local input queues that are continuously filled by the broadcasted updates. The lower the workload of the cluster is, the better the performance of PDBREP becomes with respect to CRM. The gain in query evaluation time is between 30% and 65% in average for 75% and 50% loaded clusters, respectively. The improvement goes up to 76% for queries requesting freshest data in 50% loaded cluster. This gain results from the introduction of propagation transactions besides the fast access to the local input queues instead of accessing the global log. Nevertheless, note that update propa-

gation transactions always compete with queries to achieve locks on data items. When the cluster is idle, updates are free to propagate and they can maintain the freshness of the cluster to some degree. Even if propagation transactions wait for achieving locks, propagation queues can still be maintained by broadcasted updates. By this way, refresh transactions, which are modeled as bulked sequence of propagation transactions, can use local data in input queues to perform at least some part of their task. This heavily reduces the amount of data requested by refresh transactions from global log.

PDBREP provides us with some nice characteristics. Most importantly, it allows queries to be executed in similar time measures independent of workload. As Figure 3 shows, the execution times of queries for the three workloads slightly differ from each other with PDBREP. When the workload gets lower, then the queries get bit slower because of increased number of updates that has to be propagated for high freshness requirements. Although propagation transactions have more time to proceed when workload is low, they are still slower than refresh transactions due to more commit processing. However, propagation transactions can still keep nodes fresh enough for queries with lower freshness requirements. Especially for a 50% loaded cluster, queries have better chance to find a component that satisfies a lower degree of freshness, e.g., 0.6.

Figure 4 shows how relatively long the refresh transactions are as compared to the queries that invoke them for different workloads and freshness degrees. For instance, for a 50% loaded cluster and a freshness degree of 1.0, the refresh transactions of CRM comprises 67% of query evaluation time while it is only 42% for PDBREP.

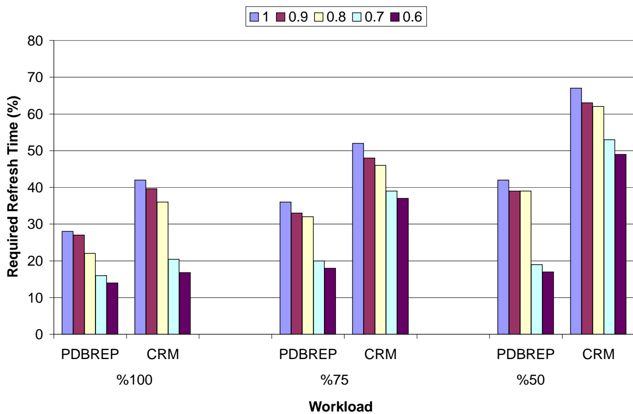

Figure 4. Impact of Refresh Transactions on Query Evaluation Times

Generally, queries spend less time for refreshing with PDBREP as compared to CRM. The reason is two fold. First, when PDBREP needs to refresh, it exploits local input queues as much as it can and then, if still required, it finally goes to the global log. On the other hand, CRM always goes to the global log to fetch changes to bring the site up to the required freshness degree. Second, with PDBREP, ongoing propagation transactions have also an impact on reducing the size of refresh transactions when the site is idle. When the cluster is fully utilized, propagation transactions are of not much use since they arrive at nodes that are already refreshed in the mean time by refresh transactions. However, for a fully loaded cluster, PDBREP has still the advantage of accessing changes localized by broadcasts.

Next, we investigate the overall performance and scalability of PDBREP with various cluster sizes from 4 to 64 nodes. We experimented with full workload of queries requesting fresh data. To show scalability, we have doubled the number of querying clients as we have doubled the size of the cluster. Note that this workload is quite high, because we wanted to explore the limits of PDBREP. The results in Figure 5 show that at least up to 64 nodes, PDBREP scales linearly with the cluster size.

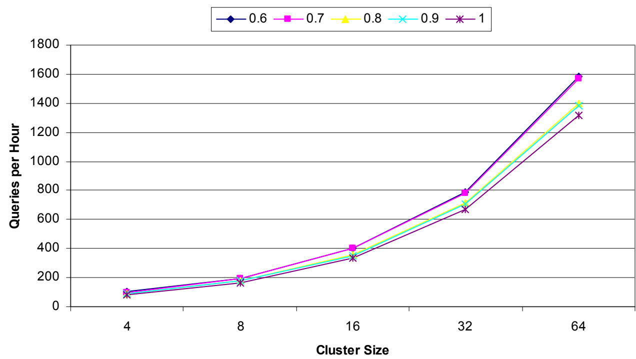

Figure 5. Query Throughput for Varying Cluster Sizes

To sum up, our PDBREP protocol on the one hand generalizes the CRM approach, such that the latter is just one setting of PDBREP, and on other the hand it generally provides a better performance.

5 Conclusions

Although replication management has come of age today, efficient replication protocols for massive numbers of replicas have still been an open question when combined OLTP and OLAP workloads must be supported, i.e., when OLAP queries shall be able to work on up-to-date data. As we have motivated in this paper, existing protocols have several drawbacks which we overcome with our new approach to lazy replication management.

The proposed protocol PDBREP provides strict freshness and correctness guarantees. It allows read-only transactions to run at several sites in parallel and deploys a sophisticated version control mechanism to ensure one copy serializable executions. Moreover, PDBREP does not require data to be

fully replicated on all sites. It works with arbitrary physical data organizations such as partitioning and supports different replication granularities. In this way, the cluster of database systems for instance can better be customized for given workloads, and thereby increase the overall performance of the system. For example, a subset of cluster nodes can be designed for certain query types.

Although PDBREP is a lazy replication protocol, it also provides access to fresh data by means of decoupled update propagation and refresh transactions. In this way, it combines the performance benefits of lazy protocols with the up-to-dateness of synchronous approaches. In addition, PDBREP also extends the notion of freshness to finer granules of data, and thereby reduces the needed refreshment efforts. Finally, PDBREP does not require a centralized component for coordination apart of a global log where the update sites place their committed update transactions.

Our experiments revealed that PDBREP is clearly superior to related previous approaches with respect to architectural as well as performance characteristics.

Acknowledgements: This research is partially supported by Microsoft Research.

References

[1] D. Agrawal and A. El Abbadi. Constrained Shared Locks for Increasing Concurrency in Databases. *Journal of Computer and System Sciences*, 51(1):53–63, Aug. 1995.

[2] F. Akal, K. Böhm, and H.-J. Schek. OLAP Query Evaluation in a Database Cluster: A Performance Study on Intra-Query Parallelism. In *Advances in Databases and Information Systems, 6th East European Conference, ADBIS 2002, Bratislava, Slovakia, September 8-11, 2002, Proceedings*, volume 2435 of *Lecture Notes in Computer Science*, pages 218–231. Springer, 2002.

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

[4] Y. Breitbart, R. Komondoor, R. Rastogi, S. Seshadri, and A. Silberschatz. Update Propagation Protocols For Replicated Databases. In A. Delis, C. Faloutsos, and S. Ghandeharizadeh, editors, *Proceedings ACM SIGMOD International Conference on Management of Data (SIGMOD'99), June 1-3, 1999, Philadelphia, Pennsylvania, USA*, pages 97–108. ACM Press, 1999.

[5] Y. Breitbart and H. F. Korth. Replication and Consistency: Being Lazy Helps Sometimes. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 12-14, 1997, Tucson, Arizona*, pages 173–184. ACM Press, 1997.

[6] P. Chundi, D. J. Rosenkrantz, and S. S. Ravi. Deferred Updates and Data Placement in Distributed Databases. In S. Y. W. Su, editor, *Proceedings of the Twelfth International Conference on Data Engineering, February 26 - March 1, 1996, New Orleans, Louisiana*, pages 469–476. IEEE Computer Society, 1996.

[7] J. Gray, P. Helland, P. O'Neill, and D. Shasha. The Dangers of Replication and a Solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada*, pages 173–182, 1996.

[8] B. Kemme and G. Alonso. Don't Be Lazy, Be Consistent: Postgres-R, A New Way to Implement Database Replication. In *Proceedings of the 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 134–143, 2000.

[9] E. Pacitti, P. Minet, and E. Simon. Fast Algorithms for Maintasining Replica Consistency in Lazy Master Replicated-Databases. In M. P. Atkinson, M. E. Orlowska, P. Valduriez, S. B. Zdonik, and M. L. Brodie, editors, *Proceedings of the 25th International Conference on Very Large Data Bases, VLDB'99, September 7–10, 1999, Edinburgh, Scotland, UK*, pages 126–137. Morgan Kaufmann Publishers, 1999.

[10] E. Pacitti, M. T. Özsu, and C. Coulon. Preventive Multimaster Replication in a Cluster of Autonomous Databases. In *Proceeding. of the 9th International Euro-Par Conference, Klagenfurt, Austria, August 26-29, 2003*, pages 318–327, 2003.

[11] E. Pacitti, E. Simon, and R. N. Melo. Improving Data Freshness in Lazy Master Schemes. In *Proceedings of the 18th International Conferenc eon Distributed Computing Systems, 26–29 May, 1998, Amsterdam, The Netherlands*, pages 164–171. IEEE Computer Society Press, 1998.

[12] C. H. Papadimitriou. *The Theory of Database Concurrency Control*. Computer Science Press, 1986.

[13] U. Röhm, K. Böhm, and H.-J. Schek. Cache-Aware Query Routing in a Cluster of Databases. In *Proceedings of the 17th International Conference on Data Engineering (ICDE2001), April 2-6, 2001 Heidelberg, Germany*, pages 641–650. IEEE Computer Society, 2001.

[14] U. Röhm, K. Böhm, H.-J. Schek, and H. Schuldt. Freshness Aware Scheduling in a Cluster of Databases. In *Proceedings of 28rd International Conference on Very Large Data Bases (VLDB2002), August 2002, Hongkong, China*, pages 754–765. Morgan Kaufmann Publishers, 2002.

[15] I. Stanoi, D. Agrawal, and A. El Abbadi. Using Broadcast Primitives In Replicated Databases. In *Proceedings of the 18th International Conference on Distributed Computing Systems, 26–29 May, 1998, Amsterdam, The Netherlands*, pages 148–155. IEEE Computer Society Press, 1998.

[16] The PowerDB Project. http://www.dbs.ethz.ch/~powerdb.

[17] Transaction Processing Council (TPC). *TPC Benchmark R (Decision Support)*. http://www.tpc.org/tpcr/spec/tpcr_current.pdf.