Fine-Grained Replication and Scheduling with Freshness and Correctness Guarantees

Fuat Akal[1], Can Türker[1], Hans-Jörg Schek[1], Yuri Breitbart[2], Torsten Grabs[3], Lourens Veen[4]

[1]ETH Zurich, Institute of Information Systems, 8092 Zurich, Switzerland, {akal,tuerker,schek}@inf.ethz.ch
[2]Kent State University, Department of Computer Science, Kent OH 44240, USA, yuri@cs.kent.edu
[3] Microsoft Corp., One Microsoft Way, Redmond, WA 98052, USA, torsteng@microsoft.com
[4] University of Twente, 7500 AE Enschede, The Netherlands, l.e.veen@student.utwente.nl

Abstract

Lazy replication protocols provide good scalability properties 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 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 update 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 moreover generalizes the notion of *freshness* to finer data granules than entire databases.

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 object 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 synchronizes *all* copies of an object within the same database transaction [3]. However, conventional eager replication protocols have significant drawbacks regarding performance and scalability [8, 11, 22], which are due to the high communication overhead among the replicas and the high probability of deadlocks. Newer eager replication protocols, such as proposed in [12, 13], try to reduce these drawbacks by using group communication. *Lazy* replication management, on the other hand, decouples replica maintenance from the "original" database transaction [5, 6, 15]. 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, additional efforts are necessary to guarantee serializable executions. Previous work on lazy replication like [4, 5] has focused 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 by conventional lazy replication techniques.

Recently, [18] addresses this issue by allowing read-only clients to define *freshness requirements* stating how up-to-date data shall be accessed. However, the approach of [18] suffers from several important shortcomings. First, it relies on full replication at a granularity of complete databases. Clearly, this precludes more sophisticated physical data organization schemes such as partial replication, partitioning or striping across sites, which can be beneficial for parallelization of queries. Second, it assumes that the transaction executes entirely at its initiation site, which may not be the case in practical database settings where the data is distributed over various cluster nodes. Third, it requires a centralized coordination component for the entire scheduling and bookkeeping, which is a potential bottleneck and single point of failure.

The objective of this paper is to attack the aforementioned shortcomings and to present a new protocol that

* This work has been supported partially by Microsoft.

Proceedings of the 31st VLDB Conference, Trondheim, Norway, 2005

covers the following requirements jointly: (1) combining the advantages of lazy and eager replica maintenance to ensure correct and efficient executions, (2) supporting arbitrary physical data organization schemes, (3) allowing users to specify freshness requirements, and (4) executing read-only transactions at several data sites in parallel. This goal cannot be achieved by simply extending previous lazy replication protocols. As the following example shows, for instance, lazy replication as proposed in [4] fails already if one allows a transaction to read objects from several sites.

Example 1: Assume that for each object there is a single primary site responsible for updates to the object. Let there be 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

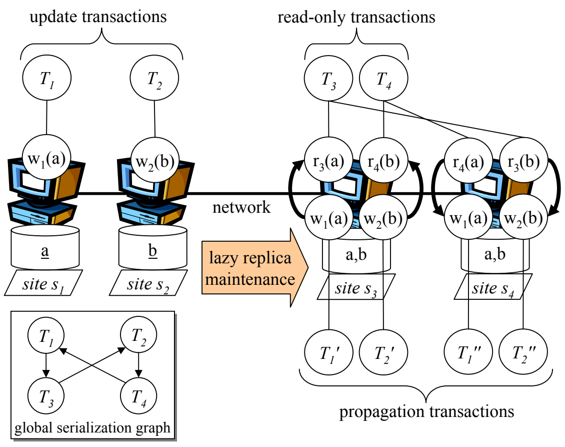

Figure 1: Lazy replication management

contain objects 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 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 and, furthermore, each site-specific schedule is locally correct.

In this paper, we present a new replication protocol, called PDBREP[1], which covers all aforementioned requirements. PDBREP exploits two important characteristics: (1) distinction between *read-only* and *update transactions* and (2) partitioning of the sites into *read-only* and *update sites* to process read-only and update transactions, respectively. The main idea of PDBREP is to exploit distributed versioning together with *freshness locking* to guarantee efficient replica maintenance that provides consistent executions of read-only transactions. Globally correct execution of update transactions over update sites is already covered by previous work, e.g. [4]. It is therefore not the concern of this paper.

Our main contributions are as follows:

- PDBREP supports different physical data organization schemes ranging from full replication at the granularity of complete databases to partial replication combined with partitioning and striping.
- PDBREP respects user-demanded freshness requirements. This includes the special case where users always want to work with up-to-date data.
- PDBREP produces correct, i.e., one-copy serializable, global executions and allows distributed executions of read-only transactions.
- We implemented PDBREP and evaluated it in various settings to reveal its performance characteristics.

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 experimental evaluation results. Section 5 talks about related work. Section 6 concludes the paper.

2 System Model

Figure 2 illustrates the model of our system. We consider a replicated database that contains objects which are distributed and replicated among the sites. Since we use relational database systems, objects are (partitions of) relations while operations are queries or data manipulation statements. For each object d_i there is a *primary site* denoted by $p(d_i)$. If site s contains a replica of d_i and $s \neq p(d_i)$, we call a replica of d_i 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.

Following previous work [5, 4], updates of an object 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 object can be ordered according to the order of their execution at the primary site. Similarly to [18], we partition all

[1]PDBREP stands for the replication protocol we implemented within Microsoft supported project PowerDB at ETH Zurich [20]

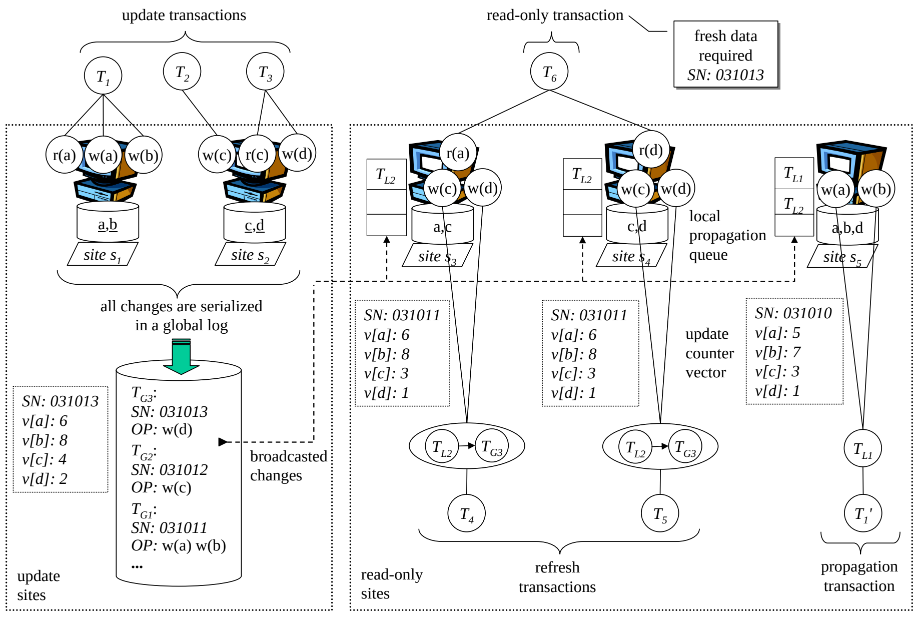

Figure 2: System architecture underlying PDBREP

sites into two classes: (1) read-only and (2) update. Read-only transactions only run at *read-only sites*, while update transactions only run at *update sites*.

Update transactions consists of at least one write operation. A write operation is any SQL-DML statement (insert, update and delete). The changes of update transactions that occur at update sites are serialized [4] and logged in a *global log*. For instance, T_{G1} represents the changes of update transaction T_1. These changes are continuously broadcasted to all read-only sites in the system and are enqueued in the local propagation queues which have the same structure as the global log. For example, T_{L1} corresponds to localized changes of update transaction T_1. The broadcast is assumed to be reliable and preserves the global FIFO order, i.e., changes are received by all read-only sites in the order they have been broadcasted by the global log component.

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 an object a if T is initiated at the primary site of a. T, however, may read any object at this site.

Example 2: Figure 2 depicts three update transactions T_1, T_2, and T_3 which only run on update sites and their write operations occur at primary sites.

Read-only transactions in turn may be initiated at any read-only site. These transactions read objects only from read-only sites. Their read operations may run at different read-only sites. This is an important generalization of [18] 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.

Example 3: Figure 2 shows a read-only transaction T_6. As the figure illustrates, read-only transactions only run on read-only sites and may distribute their operations over several sites – depending on the physical data organization or the workload situation at read-only sites.

Propagation transactions are performed during the idle time of a site in order to propagate the changes present in the local propagation queues to the secondary copies. Therefore, propagation transactions are continuously scheduled as long as there is no running read or refresh transaction. By virtue of our model, the propagation transactions for the same object are initiated from the same primary site. As a result, all updates at secondary sites of the same objects are ordered by the order of the primary transaction that performed such an update at the primary site of the object.

Example 4: Figure 2 shows the propagation transaction T_1' applying the changes of the update transaction T_1 to site s_5. 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 freshness level specified by a read-only transaction. A refresh transaction aggregates one or several propagation transactions into a single bulk transaction. 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. A refresh transaction first checks the local propagation queue to see whether all write operations up to the required timestamp are already there. If yes, it fetches these write operations from the local propagation queue and applies them to the database in a single bulk transaction. Otherwise, it retrieves whatever available in the local propagation queue and goes to the global log for the remaining part.

Example 5: Figure 2 shows two refresh transactions T_4 and T_5. They bring sites s_3 and s_4 to a state that is needed to run the read-only transaction T_6, which requires fresh data with at least a timestamp of 031013. The timestamps of sites 3 and 4 are equal to 031011. The system hence checks the local propagation queues for potentially missing changes – in the figure these are the changes made by T_2 and shown as T_{L2}. At this point, the system realizes that the changes of update transaction T_3 have not been enqueued in the propagation queue at neither sites 3 and 4 yet. So, it retrieves T_{G3} from the global log. Altogether, the refresh transaction includes T_{L2} and T_{G3}. After completing the refresh, the read-only transaction T_6 is executed.

Update, propagation, and refresh transactions are executed as decoupled database transactions. We assume that each site ensures locally correct transaction executions. We adopt the criterion *one-copy-serializability* [3] to guarantee global correctness. Hence, 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.

3 The PDBREP Protocol

3.1 Overview of the Protocol

PDBREP exploits the read-only site's idle time by continuously scheduling propagation transactions as update transactions at the update sites commit. The rationale behind this is to keep the secondary copies at the read-only sites as much as possible up-to-date such that the work of refresh transactions (whenever needed) is reduced and thus the performance of the overall system is increased.

We assume that a globally serializable schedule is produced for all update sites by some algorithm that is of no concern for this paper. Moreover, the update transactions' serialization order is their commit order. Thus, each propagation transaction inherits the number in which the 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 objects it reads at its start. PDBREP foresees two different approaches to determine this version. The *implicit* approach determines the versions of the objects accessed by a read-only transaction T from the versions of the objects at the sites accessed by T. With the *explicit* approach, users may specify the *freshness* of the data accessed by their read-only transactions as a quality of service parameter. However, explicitly specified freshness can be changed implicitly, if none of the objects involved in transaction satisfies required freshness level or if at least one of them is fresher than this level. If a read-only transaction is scheduled to a site that does not yet fulfill the freshness requirements, a refresh transaction updates the objects at that site.

With either of the two aforementioned approaches, freshness locks are placed on the objects at the read-only sites to ensure that ongoing replica maintenance transactions do not overwrite versions which are still needed by ongoing read-only transactions. A *freshness lock* thus represents a barrier that disallows updates of an object beyond a given freshness level. Freshness locks keep the objects accessed by a read-only transaction at a certain freshness level during the execution of that transaction. When a read-only transaction T_i with the freshness requirement of TS wants to access some objects, it first has to acquire freshness locks on these objects. The procedure for acquiring freshness locks is performed in two steps:

1. T_i asks for a freshness lock on the object with the timestamp TS. The freshness locking procedure places a lock with the timestamp TS if the current freshness level of the object is not younger than TS. Otherwise, it places the lock with the timestamp of the current freshness level of that object. In the following, we will use the term *pre-lock* to refer to a lock placed in the first phase of the freshness locking procedure.

2. Depending on the current freshness level of the various objects, the pre-locks can differ with respect to the freshness level. To ensure that the transaction T_i will read consistent data, all freshness locks of T_i must be brought up to the same freshness level (if this is not already the case). That is, freshness locks are upgraded to timestamp of either the freshest site in the system or the youngest object pre-locked by T_i.

Example 6: Figure 3 shows freshness locking in action for three possible cases when a read-only transaction T_1 wants to access the objects a and b (residing on different sites) with the freshness requirement ts_3:

1. The left diagram depicts the case where none of the involved sites satisfies the freshness requirement of T_1, which demands the freshest data. Therefore, freshness locks are placed on both objects with the timestamp ts_3, which is assumed to be the freshest in the example. Thus, the version of data is determined by the read-only transaction explicitly.

2. The middle diagram shows the case where both involved sites have younger data than required. This time the locks are first placed on the objects with corresponding current freshness level, i.e., a is locked with ts_5 and b is locked with ts_4. Then, the freshness lock on b is upgraded to ts_5.

3. The right diagram exhibits the case where some objects are older than required while others satisfy the freshness requirement of T_1. In case of a "fresh enough" data, the lock is set to the current freshness level of that item. In the example, a is therefore locked with ts_5. "Not fresh enough" data is locked with the required timestamp of the transaction. Hence, b is locked with ts_3. This lock is then upgraded to ts_5 in the second phase of freshness locking.

For the last two cases, note that the actual version of data accessed is implicitly determined by the freshness locking algorithm.

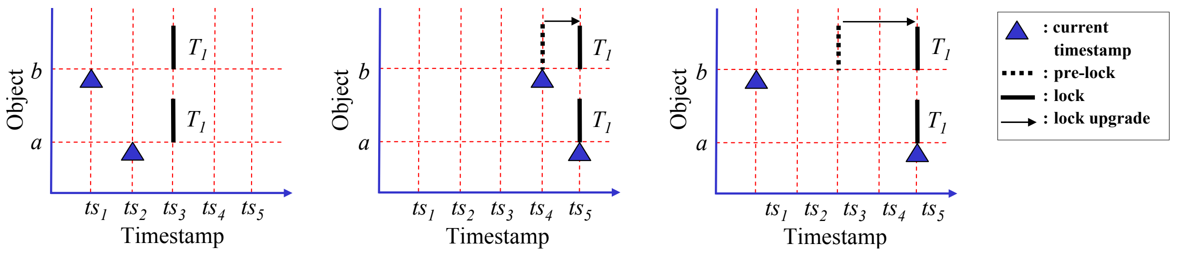

Figure 3: Freshness locking for $T_1 = r_1(a)r_1(b)$ with $TS = ts_3$

The freshness locking procedure of PDBREP guarantees that transactions always operate on transactionally consistent versions. PDBREP ensures one-copy serializable executions of propagation, refresh, and read-only transactions.

3.2 Definition of the Protocol

After having 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 transactions in the order of their sequence numbers does not guarantee by itself one-copy serializability. Therefore, we need an additional data structure.

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 maintain an* update counter vector *v of dimension n defined as follows. For any object a stored at site s, $v(s)[a]$ 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 the 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. This is to ensure that distributed read-only transactions always read consistent versions of objects. Moreover, vectors v may differ between sites while propagation transactions are ongoing. Figure 2 illustrates these vectors together with the site number. Initially, all vector components at all sites are zero. From a conceptual point of view and for ease of explanations we have these vectors. But, the implementation would be different. For instance, we would use (dynamic) hash tables in order to be extensible once we will change the partitions over time.

Definition 3 (Version Vector of a Read-only Transaction) *The* version vector *of transaction T is similar to the update counter vectors of sites and denoted as $v(T)$. It determines the version of data that will be read by T.*[2]

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

Definition 4 (Data Freshness) Freshness *is defined by means of timestamps. The younger the timestamp, the fresher the data is.*

We assume that there exists a function that maps the *freshness requirement* of a read-only transaction to a corresponding timestamp. We give an example of such a function in Section 4.

Definition 5 (Required Freshness Level) *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. 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 level of the freshest site or of the global log if none of the sites satisfy the freshness level required by T. If $TS_T \leq TS_s$ holds, then the refresh transaction R is empty.

A refresh transaction is thus 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 prevent propagation and refresh transactions that bring an object to a freshness level that is above of the freshness level of an ongoing read-only transaction.

Definition 6 (Freshness Lock) *A* freshness lock *is placed on an object a at site s with an associated freshness timestamp f demanded by the acquiring read-only transaction.*

[2]To ease the comparison between the vector of a read-only site and the vector of a transaction, the version vector of a transaction contains an element for all objects present in the system. Indeed, we could reduce the size of the transaction vector to the number of locked objects.

Such a lock disallows a write operation on object a on site s if this operation brings object a on site s to a younger timestamp than f.

This definition states that locking is done at the granularity of objects. Since we do not restrict the granularity of the object, the locking granularity could be as large as a whole database and as small as (a partition of) a relation.

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

The scheduling of a read-only transaction is performed according to Algorithm 1 which consists of three building blocks that implement the following rules based on the previous definitions:

Rule 1: Each read-only transaction submitted at site s requests freshness locks with its required freshness timestamp at s for all objects read by the transaction. If an object does not exist at s, then the freshness lock is requested from a site that has a copy of the object. The transaction waits for the freshness locks to be granted. Note that *not all* copies of an object have to be freshness locked, but *only one*.

Rule 2: Once all locks are acquired, the version vector and the freshness timestamp of that transaction are determined. If at least one of the sites involved in the transaction is fresh enough to satisfy the transaction's requirement, the counter vector of the site with the highest timestamp and its timestamp are assigned as the transaction's version vector and timestamp, respectively. Otherwise, the counter vector and the timestamp of the global log are used to enforce accessing up-to-date data.

Rule 3: A read-only transaction T_j submits its read operation $r_j(a)$ to a previously locked read-only site s_k that stores a copy of a. If $r_j(a)$ is submitted to such a site s_k, then the following rules apply:

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

2. If $v(s_k)[a] < v(T_j)[a]$, then the freshness lock which was granted to T_j on object a is upgraded to the freshness timestamp of T_j. Upgrading a freshness lock implicitly invokes a refresh transaction and delays the read operation until $v(s_k)[a] = v(T_j)[a]$ holds. Then, the read operation is processed on s_k.

Since all locks are preclaimed, the case $v(s_k)[a] > v(T_j)[a]$ cannot happen.

Rule 4: A read-only transaction has to release all its acquired freshness locks with its termination.

As we stated earlier, PDBREP continuously deploys propagation transactions to exploit the idle time at the sites for propagating changes. To avoid interference with execution of a read or a refresh transaction on a site s, propagation transactions are not processed on s when there is a read-only transaction scheduled or running. The refresh transactions, on the other hand, are demanded by read-only transactions. When a read-only transaction acquires a freshness lock on an object which is not fresh enough, PDBREP automatically runs a refresh transaction to bring that object to demanded freshness level. Algorithm 2, 3 and 4 perform the continuous scheduling and execution of propagation and refresh transactions according to the following rules:

Algorithm 1: Scheduling read-only transactions

```
Data: read-only transaction $T_i = \{op_i\}$,
      freshness timestamp $ft$ (0 = implicit approach);
// place freshness locks
foreach $op \in \{op_i\}$ do
    let $a$ denote the object read by $op$;
    acquire lock on $a$ at some site $s_j$ storing a copy of $a$;
    $S := S \cup s_j$;
end
// compute transaction's version vector and timestamp
if $ft > max\left(ts_{s_j} \mid s_j \in S\right)$ then
    // none of the involved sites meets required freshness
    // use global version vector to access the freshest data
    $v(T) = v(s_{gl})$;
    $ts = timestamp(s_{gl})$;
else
    // version vector of the freshest site involved in $T_i$
    $v(T) = max\,(v(s_l) \mid s_l \in S)$;
    $ts = timestamp(s_l \mid v(s_l) = v(T))$;
end
// execute the operations
foreach $op \in \{op_i\}$ do
    let $s_j$ denote the site $op$ is routed to;
    let $a$ denote again the object $op$ reads;
    BOT;
    if $v(s_j)[a] < v(T)[a]$ then
        upgrade $a$'s freshness lock at site $s_j$
        to the required timestamp $ts$;
        wait until $s_j$ is brought up to $ts$ level;
    end
    process $op$ at $s_j$;
    release $T$'s freshness locks at site $s_j$;
    EOT;
end
```

Rule 5: Propagation and refresh transactions execute changes in the order of the sequence numbers.

Rule 6: A write operation $w_n(a)$ of a propagation or refresh transaction submitted at site s_k is handled as follows:

1. Let T be a read-only transaction holding the "oldest" freshness lock on object a at site s_k. By "oldest" we mean that there is no other freshness lock on the same object at the same site which refers to an older freshness timestamp. If there is no freshness lock on the object a at site s_k (in case of propagation transaction) or $v(s_k)[a] < v(T)[a]$ holds, then $w_n(a)$ is executed and corresponding update counter $v(s_k)[a]$ is incremented by one (even if object a is not stored at s_k).

2. Otherwise, the operation is delayed until the conflicting freshness locks are released at the site.

Algorithm 2: Scheduling Propagation and Refresh Transactions

```
Data: size of bulked propagation transaction $propSize$,
      timestamp of last write operation at site $s_i$ $SN(s_i)$,
      local propagation queue at site $s_i$ $lpq$
while $true$ do
    if there is no scheduled or running read-only transaction
    then
        if sizeOf($lpq$) $\geq propSize$ then
            // enough changes enqueued for bulk propagation
            run propagation transaction($propSize$);
        end
    else
        // get the user demanded freshness timestamp
        let $requiredTS$ be the minimum timestamp on which
        there is a freshness lock at site $s_i$;
        if $requiredTS > SN(s_i)$ then
            // $s_i$ does not satisfy user's freshness demand
            run refresh transaction($requiredTS$);
        end
    end
end
```

Rule 7: A propagation or refresh transaction T submitted at site s_k has to overwrite the site's timestamp SN_{s_k} with its commit by the value of SN_T.

As Algorithm 2 shows, PDBREP exploits bulk transactions for applying changes to the read-only databases. While propagation transactions use predefined and relatively small bulk sizes, refresh transactions try to perform all required changes using one (sometimes large) bulked transaction. In contrast to a propagation transaction, a refresh transaction does not know its bulk size in the first place. It applies all changes occurred up to the demanded timestamp to the database. If all required changes are already localized in the local propagation queue, it retrieves and applies them. Otherwise, it gets what is available in the propagation queue and goes to the global log for the remaining part.

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.

3.3 Discussion

For executions of read-only and propagation transactions at read-only sites to be correct, the global schedule must be serializable. As stated before, we 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.

Theorem 1 (Serializability of PDBREP schedules) PDBREP *produces one-copy serializable schedules for read-only and propagation transaction on read-only sites.*

Algorithm 3: Propagation Transaction

```
Data: bulk propagation transaction $P$ with operations $\{op_j\}$,
      number of operations in $P$ $propSize$, site $s_i$, time-
      stamp of last committed write operation at $s_i$ $SN(s_i)$
BOT;
// generate fixed size bulk propagation transaction
retrieve next $propSize$ write operations from local input
queue into $P$;
let $SN(P)$ be the timestamp of the last write operation in $P$;
// execute the (write) operations
foreach $op \in \{op_j\}$ do
    let $a$ denote the object $op$ writes;
    process $op$;
    // update counter of object $a$ in $s_i$'s version vector
    $v(s_i)[a] := v(s_i)[a] + 1$;
end
remove all operations in $P$ from local input queue;
// set $SN(s_i)$ to timestamp of last executed write operation
$SN(s_i) := SN(P)$;
EOT;
```

Due to space restrictions we skip the formal proof of this theorem which can be found in [1]. Instead, we discuss some other nice characteristics of PDBREP.

Avoiding global deadlocks. With PDBREP, 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.

Lemma 1 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. For the full formal proof of this lemma, we again refer to [1].

Effect of Refresh Transactions. 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 at site S, we distinguish two cases:

1. If timestamp of P is less than or equal to timestamp of S, then P is simply skipped.

Algorithm 4: Refresh Transaction

```
Data: bulk refresh transaction $R$ with operations $\{op_j\}$,
      read-only transaction $T$ that invokes $R$,
      required freshness timestamp $requiredTS$,
      timestamp of last committed write operation at site $s_i$
      $SN(s_i)$
BOT;
// generate bulk refresh transaction
retrieve all write operations with timestamp values less than
or equal to $requiredTS$ from local input queue into $R$;
let $SN(R)$ be the timestamp of latest write operation in $R$
performed at $s_i$;
if $requiredTS > SN(R)$ then
    // all required write operations are not yet
    // available in the local queue
    retrieve and append all write operations with timestamp
    values between $SN(s_i)$ and $requiredTS$ from global log
    into $R$;
end
// execute the (write) operations
foreach $op \in \{op_j\}$ do
    let $a$ denote the object $op$ writes;
    process $op$;
    // update counter of object $a$ in $s_i$'s version vector
    $v(s_i)[a] := v(s_i)[a] + 1$;
    if $v(s_i)[a] = v(T)[a]$ then
        // object is already at required freshness level
        // rest of refresh transaction is hence not necessary
        exit foreach loop;
    end
end
remove all operations in $R$ from local input queue;
// set $s_i$'s timestamp to that of the last executed
// write operation
$SN(s_i) := SN(R)$;
EOT;
```

2. Otherwise, P waits until T finishes.

From this we conclude that no additional dependencies are introduced.

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 objects 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 implemented a prototype on which we ran 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 update node, a global log, a distributed coordinator layer and a client simulator. The evaluation has been conducted on a cluster 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.

We evaluated PDBREP for three different settings where we switch on and off the continuous broadcasting and propagation features:

Setting 1: No Broadcasting and No Propagation

Setting 2: Broadcasting and No Propagation

Setting 3: Broadcasting and Propagation

We used the database and queries of the TPC-R benchmark [21]. We created the TPC-R database by using scale factor 1.0 which resulted in a database of roughly 4.3GB together with indexes that we optimized with Microsoft Index Tuning Wizard. We divided our cluster into node groups of four nodes. Each node group had the full replica of whole database, e.g., for 64 node cluster, we had 16 replicas (node groups). Within the node groups, we fully replicated the relatively small tables (e.g. Nations, Region) and hash partitioned the huge ones (e.g. Orders, LineItem). We implemented update counter vectors as an array of integers. In the experiments, we submitted queries to node groups where they were evaluated on four nodes in parallel. In another words, the number of sites involved in a distributed read transaction was four. For routing and load balancing, we used the *Balance-Query-Number* algorithm.

The global log broadcasts the changes of update transactions as bulk packages of a certain size. In the experiments, we used a bulk size that is equal to the number of updates per second. We set the update rate at 200 updates per second, a relatively high value compared to, e.g., [18]. Propagation and read-only transactions run at the read-only sites. Refresh transactions also run at the read-only sites except that they request data from the global log when the data is not available locally.

In our experiments, we varied three parameters:

- Freshness requirement [0.6, 0.7, 0.8, 0.9, 1.0]
- Workload [50%, 75%, 100%]
- Propagation bulk size [200, 400, 800, 1600]

The actual timestamp at which a lock will be requested for a read-only transaction is determined as follows. We use a freshness window of n seconds, and map a specified freshness requirement $fl \in [0.0, 1.0]$ to that window using the function $f(fl) = n * (1 - fl)$. In our experiments, we used 30-second windows. That is, data with freshness 1.0 is current, while data with freshness 0.0 is 30 seconds old.

This provides a definition of freshness that is independent of the total system runtime as well as of the update rate. We varied the required freshness between 0.6 and 1.0, thus requesting data that was at most 12 seconds old.

Workload is defined as the percentage of available processing time the cluster spends executing read-only and refresh transactions. Thus, if a client supplies read-only transactions at half the rate the cluster can execute them, the workload will be 50%. Our test client dynamically controls the rate of read-only transactions to keep the workload at the preset value.

As [18] has shown, applying refresh transactions in bulk rather than executing them individually significantly improves performance. Since propagation transactions also apply changes to the database, we bulk them, too. This introduces some latency into the system, the effects of which we studied for various bulk sizes.

4.2 Results

Experiment 1: Our first experiment evaluates the influence of broadcasting and propagating updates on cluster performance. We performed this experiment for various workloads and freshness requirements to measure the impact of these features under different conditions. The results are depicted in Figure 4 and Table 1.

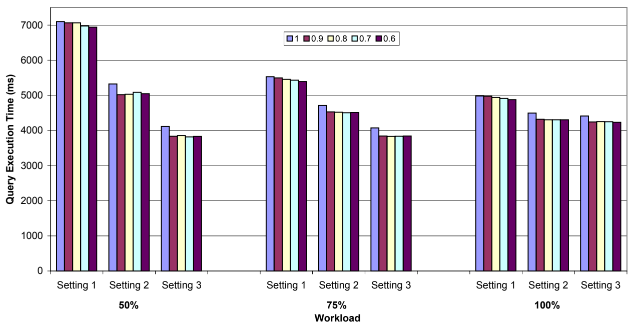

Figure 4: Average Query Evaluation Times for Different Workloads and Freshness Values

Setting	Freshness 0.6 - 0.9			Freshness 1.0		
	50%	75%	100%	50%	75%	100%
1	100%	100%	100%	100%	100%	100%
2	72%	83%	87%	75%	85%	90%
3	55%	70%	86%	58%	74%	89%

Table 1: Relative Query Execution Time for Various Workloads w.r.t. Setting 1

The results clearly show the benefit of using broadcasting and propagation. For instance, in case of a workload of 50% we achieve performance improvements up to 82%. We see that turning on propagation and/or broadcasting always improves performance. The lower the workload is, the higher the gain in performance becomes.

Looking at the average size of the refresh transactions, as depicted in Figure 5, we see that propagation effectively eliminates the need for refresh transactions except for the maximum freshness requirements and workloads. This results in query execution times that are practically independent of the overall workload for the given update rate.

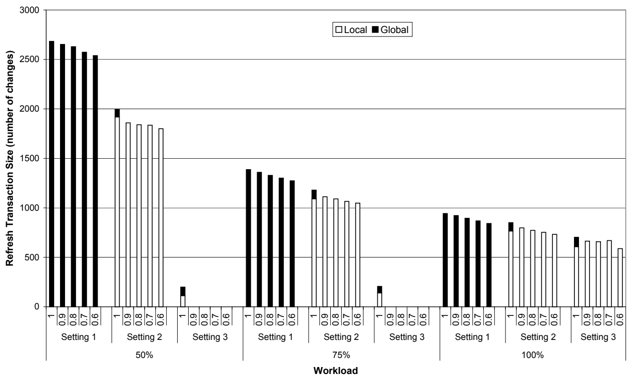

Figure 5: Average Refresh Transactions Size for Different Workloads and Freshness Values

At high workloads, there is less time for propagating changes between read-only transactions. At 100% load, there is virtually no time at all for propagation. There is just some processing overhead at the beginning and end of a read-only transaction, during which a propagation transaction may be executed. This explains the small performance improvement under this condition.

The effect of broadcasting may also be seen in Figure 5. When updates are broadcasted, refresh transactions may be executed completely locally. Only for the very highest freshness requirements, the updates that have not been broadcasted yet have to be fetched from the global log. At Setting 3, the total amount of fetched updates is between 3% and 10% of that at Setting 1 depending on the workload.

Figure 5 exhibits another interesting result. At first sight, one would expect that broadcasting alone would not affect the average size of refresh transactions, just the speed at which they are executed. However, the results depicted in Figure 5 clearly show a smaller average transaction size when broadcasting is enabled. The explanation for this is that we have a fixed workload. With propagation enabled, refresh transactions execute faster, which shortens the overall query execution time. At a fixed workload, this means that the query rate will be higher. This results in a shorter time interval between queries, and thus in less updates to process in between.

Last but not least, Figure 5 also shows that the global log might become a bottleneck in Setting 1. In setting 2 and 3, on the other hand, it is hardly involved in refresh transactions. Consequently, it does not cause a bottleneck.

Experiment 2: This experiment aims at determining the scalability of PDBREP. The results of the first experiment suggest that scalability should be good, especially in case of lower freshness requirements, where refresh transactions are expected to be completely locally. We ran this experiment at 50% cluster load. Figure 6 shows the results.

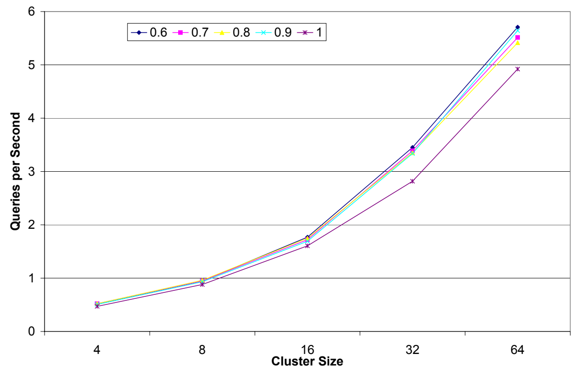

Figure 6: Query Throughput for Varying Cluster Sizes

The results for freshness 0.9 and lower are virtually identical. At freshness 1.0, throughput is slightly less. This was expected given the results of Experiment 1.

Experiment 3: As mentioned above, bulking propagation transactions introduces some latency into the system. As shown in Experiment 1, refresh transactions become necessary, which increases the query execution time. Obviously, this is also dependent on the required freshness level. The purpose of this experiment is to determine how query execution time is affected by the propagation bulk size. Figure 7 shows the query execution times for various required freshness levels and propagation bulk sizes at a workload of 50%.

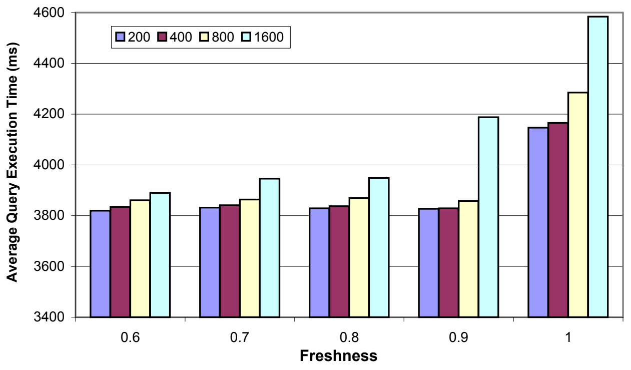

Figure 7: Effect of Propagation Bulk Size on Query Execution Time

Recall that we perform 200 updates per second, i.e., if there are no ongoing read-only transactions, propagation transactions are performed once every 1, 2, 4, and 8 seconds for bulk sizes of 200, 400, 800 and 1600, respectively. Our 30-second freshness window means that the freshness requirements 0.6 to 1.0 refer to 12, 9, 6, 3, and 0 seconds old data. The graph shows the expected result: when we require data that is older than the size of the propagation interval, this data will already have propagated and no refresh is necessary. This results in shorter execution times. When we require newer data, refresh transactions will be necessary and execution time rises. As we saw in Figure 5, at freshness 1.0 a part of the updates must be fetched from the global log. This explains the increase of maximum 10% in query execution times. It is worth noting that this essentially is the same effect except that the latency is caused by bulking broadcasts rather than by bulking propagation transactions.

Experiment 4: In this experiment, we compared fully functional PDBREP (Setting 3) with *Freshness Aware Scheduling* (FAS) introduced in [18]. PDBREP is based on a *partial replication* scheme where the database tables are divided into partitions and these partitions are replicated on different nodes. FAS in contrast relies on *full replication* where each node has an exact copy of the entire database. For the experiment, we used a 64-node cluster. To measure the performance of FAS, we fully replicated the TPC-R database on each node in the cluster. We executed each query on a single node. To measure the performance of PDBREP, we used the node group settings that we sketched in Section 4.1. The results are depicted in Figure 8.

For all three workloads, PDBREP performs significantly better than FAS. When the cluster is 100% loaded, throughput of the cluster is 30% higher in average with PDBREP as compared to FAS. The reason is two fold. First, PDBREP allows a query be evaluated in parallel on several nodes and gains from this parallelization. Second, since PDBREP uses partitioning of data while FAS relies on full replication, the size of the refresh transactions is less than for FAS. So, the refresh process takes longer with FAS. Besides, PDBREP mostly uses locally enqueued data to refresh (see also Figure 5). The lower the workload is, the better is the performance of PDBREP with respect to FAS. The gain in query throughput is 72% and 125% in average for 75% and 50% loaded clusters. This gain results from the introduction of propagation transactions as well as from the partitioned refreshment and query parallelization. That is, PDBREP not only allows finer granularity and distributed read-only transaction execution as advantages over FAS, but also performs better than FAS.

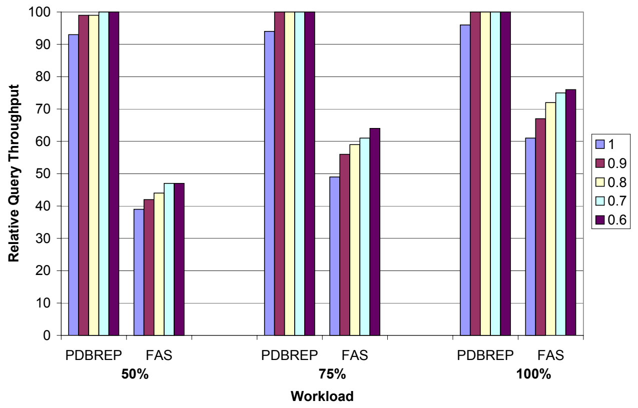

Figure 8: PDBREP vs FAS : Relative Query Throughput

To sum up, the main features of PDBREP – update broadcasting, update propagation, and freshness requirements – work together to deliver good performance. Update broadcasting improves scalability, update propagation makes query execution times nearly independent of the overall workload, and by specifying slightly lower freshness requirements we can allow the cluster to maximize the effect of these features, thus improving query execution times. Of course, we can still require up-to-date data as well.

5 Related Work

Probably the first paper that discussed many replicated relation fragments and possibilities to maintain them is in [19]. The work presented here is continuation and significant improvement of our previous work [18]. We have described the main differences already in the introduction. In a more general setting, much works has been carried out on database replication. To solve the limitations of conventional eager replication, for instance, group communication has been proposed in [13]. It first executes a transaction locally and then multicasts all changes as a write-set in a single message at the end of the transaction. Although group communication reduces the messaging overhead, it still produces a large number of message exchanges. And, it is not clear if it scales up for large clusters.

Ganymed [17] is a middleware-based replication system whose roots stem from the early phase of the PowerDB project at ETH Zurich. Similarly to PDBREP, it combines eager and lazy replication, and also distinguishes read-only and update transactions. While read-only transactions can be executed at any replica, update transactions are sent to primary site first. Then, all changes occurred in those update transactions are applied to all replicas by using write-sets. Ganymed relies on full replication and does not support freshness. Read-only transactions always see the latest version of the database.

[2] introduces distributed versioning, where a central scheduler component keeps track of the version numbers of all tables at all replicas. At the beginning of a transaction, clients inform the scheduler about the tables they will access. Then, the scheduler assigns version numbers to transactions for each table they will access. The scheduler forwards updates to all replicas and reads to a replica. Every transaction that updates a table increases its version number. All operations on a particular table are executed in version number order. This approach however does not distinguish sites as update and read-only sites. Thus, read-only transactions prevent the execution of conflicting update transactions when strict consistency is required. Hence, this approach is more likely to work on rather application scenarios where updates are rare.

Older lazy replication techniques [4, 5, 6] primarily focused on performance or correctness only. None of these techniques consider data freshness. [14] proposes a refreshment algorithm to improve freshness in lazy master replicated databases. The main principle of the algorithm is to delay refresh transactions for a certain deliver time. [15] extends this algorithm towards multi-master configurations. Neither algorithms provide access to up-to-date data. However, they keep the mean freshness degree of the data accessed by read-only transactions at high values and introduces negligible loss of freshness.

[7] generalizes the notion of strong serializability and shows how it can be implemented based on lazy replication. It distinguishes read-only and update transactions. Read-only transactions execute at replicas to which they are submitted, while update transactions execute at the primary site, where they are serialized and lazily propagated to the replicas. The changes of an update transaction, which are collected at the primary site, are broadcasted to all replicas and enqueued into local queues in their commit sequence orders. At every replica, there is a refresher process which dequeues updates and applies them to the database. In fact, there is one refresh transaction for every update transaction. A read-only transaction, on the other hand, is executed at a single site. The system does not support loose currency and the read-only transactions always access only up-to-date data. When a read-only transaction is scheduled at a replica, it first checks the sequence number of the replica. If the replica is not fresh, the read-only transaction is either forwarded to the primary site or delayed until the refresher process brings the replica to the up-to-date state.

[10] introduces MTCache, which is SQL Server's mid-tier database caching solution. The main idea of this approach is to offload some work from a backend server to intermediate servers and thus to improve system throughput and scalability. MTCache uses current support of SQL Server for transactional replication to propagate changes that occur at the backend database. The propagation is performed periodically in a single transaction by preserving the commit order. So, the applications always see a consistent but not necessarily the latest state. MTCache fits in today's storefront application scenarios which are read dominated. Recently, [9] extended MTCache to allow applications to explicitly specify their currency and consistency requirements in queries. In this model, query plans are generated according to the known currency and consistency properties of the replicas by taking into account the user's freshness demands. When local data does not satisfy the currency requirement, remote queries are generated. That is, MTCache does not take any action to bring the local data to the required level. Instead, cache maintenance continues independently of the query execution.

6 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 replication management.

The proposed protocol PDBREP supports both 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 uses lazy replication, 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 eager 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 with PDBREP provide some insights on the influence of continuous broadcasting and propagation on the overall performance. First of all, the experiments showed that PDBREP scales even with higher update rates (compared to that used in other work). Furthermore, they revealed that scenarios with lower workloads in particular significantly benefit from broadcasting and propagating updates. Another important finding is that for slightly lower freshness requirements usually refresh transactions are not necessary anymore because propagation transactions keep the sites fresh enough.

Finally, PDBREP provides the flexibility to investigate autonomic computing issues like how to dynamically adapt the cluster database design to changing parameters like the workload. This will be part of our future work.

Acknowledgements: We would like to thank the reviewers of this paper for their constructive comments.

References

[1] F. Akal, C. Türker, H.-J. Schek, T. Grabs, and Y. Breitbart. Fine-Grained Lazy Replication with Strict Freshness and Correctness Guarantees. TR 457, ETH Zürich, Department of Computer Science, Sept. 2004. `http://www.dbs.ethz.ch/publications/papers/457.pdf`.

[2] C. Amza, A. L. Cox, and W. Zwaenepoel. Distributed Versioning: Consistent Replication for Scaling Back-end Databases of Dynamic Content Web Sites. In *Proc. of the Int. Middleware Conf., June 16-20, 2003, Rio de Janeiro, Brazil*, pp. 282–304, 2003.

[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 *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pp. 97–108, 1999.

[5] Y. Breitbart and H. F. Korth. Replication and Consistency: Being Lazy Helps Sometimes. In *Proc. of the 16th ACM Symposium on Principles of Database Systems*, pp. 173–184, 1997.

[6] P. Chundi, D. J. Rosenkrantz, and S. S. Ravi. Deferred Updates and Data Placement in Distributed Databases. In *Proc. of the 12th Int. Conf. on Data Engineering*, pp. 469–476, 1996.

[7] K. Daudjee and K. Salem. Lazy Database Replication with Ordering Guarantees. In *Proc. of the 20th Int. Conf. on Data Engineering*, pp. 424–435, 2004.

[8] J. Gray, P. Helland, P. O'Neill, and D. Shasha. The Dangers of Replication and a Solution. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pp. 173–182, 1996.

[9] H. Guo, P.-A. Larson, R. Ramakrishnan, and J. Goldstein. Relaxed Currency and Consistency: How to say "Good Enough" in SQL. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pp. 815–826, 2004.

[10] H. Guo, P.-A. Larson, R. Ramakrishnan, and J. Goldstein. Support for Relaxed Currency and Consistency Constraints in MTCache. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pp. 937–938, 2004.

[11] R. Jimenez-Peris, M. Patino-Martinez, G. Alonso, and B. Kemme. Are Quorums an Alternative for Data Replication? *ACM Transaction on Database Systems*, 28(3):257–294, 2003.

[12] R. Jimenez-Peris, M. Patino-Martínez, B. Kemme, and G. Alonso. Improving the Scalability of Fault-tolerant Database Clusters. In *Proc. of the 22nd Int. Conf. on Distributed Computing Systems*, pp. 477–484. 2002.

[13] B. Kemme and G. Alonso. Don't Be Lazy, Be Consistent: Postgres-R, A New Way to Implement Database Replication. In *Proc. of the 26th Int. Conf. on Very Large Data Bases*, pp. 134–143, 2000.

[14] E. Pacitti, P. Minet, and E. Simon. Replica Consistency in Lazy Master Replicated Databases. *Distributed Parallel Databases*, 9(3):237–267, 2001.

[15] E. Pacitti, M. T. Özsu, and C. Coulon. Preventive Multi-master Replication in a Cluster of Autonomous Databases. In *Proc. of the 9th Int. Euro-Par Conf.*, pp. 318–327, 2003.

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

[17] C. Plattner and G. Alonso. Ganymed: Scalable Replication for Transactional Web Applications. In *Proc. of the 5th ACM Int. Middleware Conf.*, pp. 155–174, 2004.

[18] U. Röhm, K. Böhm, H.-J. Schek, and H. Schuldt. FAS - A Freshness-Sensitive Coordination Middleware for a Cluster of OLAP Components. In *Proc. of 28rd Int. Conf. on Very Large Data Bases*, pp. 754–765, 2002.

[19] M. Stonebraker. Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES. *IEEE Trans. Software Eng.*, 5(3):188–194, 1979.

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

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

[22] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2001.