Don't be lazy, be consistent: Postgres-R, A new way to implement Database Replication *

Bettina Kemme Gustavo Alonso
Information and Communication Systems Group, ETH Zürich, Switzerland
{kemme,alonso}@inf.ethz.ch

Abstract

Database designers often point out that eager, update everywhere replication suffers from high deadlock rates, message overhead and poor response times. In this paper, we show that these limitations can be circumvented by using a combination of known and novel techniques. Moreover, we show how the proposed solution can be incorporated into a real database system. The paper discusses the new protocols and their implementation in PostgreSQL. It also provides experimental results proving that many of the dangers and limitations of replication can be avoided by using the appropriate techniques.

1 Introduction

Existing replication protocols can be divided into *eager* and *lazy* schemes [GHOS96]. Eager protocols ensure that changes to copies happen within the transaction boundaries. That is, when a transaction commits, all copies have the same value. Lazy replication protocols propagate changes only after the transaction commits, thereby allowing copies to have different values. While eager replication emphasizes consistency, lazy replication pays more attention to efficiency.

Among database designers, there is the widespread belief that eager replication is not practical. The "dangers" of eager replication have been analyzed by Gray et al. [GHOS96] and, since the publication of those results, the research focus has shifted towards lazy replication [CRR96, PMS99, ABKW98, BKR+99]. The drawback of lazy replication is that, if consistency is necessary, many non trivial problems arise. Namely, in the case of update everywhere (each copy can be updated), maintaining consistency is usually left to the user. If only a primary copy can be updated, consistency is achieved at the price of introducing a bottleneck and a single point of failure. In addition, recent results prove that consistency can only be guaranteed when the system configuration is severely restricted.

We see these as serious limitations. The thesis defended in this paper is that if the goal is to achieve consistency and the computing environment allows it, then eager replication should be used. In spite of what is commonly assumed, eager replication is perfectly feasible in many environments. A good example are the computer clusters which can be found behind many Internet sites. However, in order to circumvent the limitations of traditional solutions, it is necessary to rethink the way transaction and replica management is done. In this paper, we demonstrate how eager replication can be implemented in practice. Some of the techniques we use include executing the transaction first locally on shadow copies and postponing the propagation of updates to the end of the transaction, using group communication primitives for preordering transactions, and acquiring all locks a transaction needs in an atomic step. To prove the feasibility of these ideas, we have implemented them in Postgres-R, an extension of PostgreSQL [Pos98] and tested them extensively. The results prove that eager replication is feasible in clusters of computers and can scale to a relatively large number of nodes.

The paper is organized as follows. Section 2 discusses related work. Section 3 explains the principle techniques of our approach and presents a basic protocol. Section 4 discusses the architecture and implementation of Postgres-R. Section 5 presents performance results. Section 6 discusses configuration management and partial replication. Section 7 concludes the paper.

*Part of this work has been funded by ETH Zürich within the DRAGON Research Project (Reg-Nr. 41-2642.5)

Proceedings of the 26th VLDB Conference, Cairo, Egypt, 2000.

2 Related Work

2.1 The dangers of replication ...

Text book eager replication protocols use update everywhere (e.g., *read-one/write-all-available*) and quorums to minimize overhead [BHG87]. With very few exceptions, these protocols have never been used in practice and Gray et al. [GHOS96] have pointed out why. These protocols coordinate each operation individually, using distributed locking and 2-phase commit. As a result, when the number of nodes increases, transaction response times, conflict probability and deadlock rates grow significantly. From these results, Gray et al. concluded that eager replication was not practical and suggested to use lazy approaches instead. Indeed, only few commercial systems implement eager replication. *Oracle Advanced Replication* provides an eager protocol which first executes an update locally and then "after row" triggers are used to synchronously propagate the changes and to lock the corresponding remote copies. Most other solutions mainly focus on availability and represent highly specialized solutions (e.g., Tandem's RDF or Informix's HDR). In general, commercial systems clearly favor lazy approaches [Sta94]. For instance, Sybase provides an extended publish-and-subscribe scheme which tries to minimize the time copies are inconsistent. As another example, IBM Data Replicator uses a pull strategy whereby a client will not see its own updates unless it requests them.

2.2 ... lazy solutions ...

On the research side, lazy replication has been studied using very different approaches like weak consistency models [PL91, KB91, GN95], economic paradigms [SAS+96] or epidemic strategies [AES97]. More recent work has explored lazy strategies that still provide consistency. Thus, Chundi et al. [CRR96] have shown that in lazy primary copy schemes, serializability cannot be guaranteed without restricting the placement of primary and secondary copies in the system. Recent work by Pacitti et al. [PMS99] and Breitbart et al. [BKR+99] has attempted to minimize this limitation. As a major drawback, all these approaches are primary copy. Furthermore, transactions cannot update data items whose primary copies reside on different sites and, in real applications (specially in clusters), the complexities and limitations on replica placement are likely to be a significant liability.
Another way to provide consistency has been to combine eager and lazy approaches. Anderson et al. [ABKW98] have proposed a system that is eager in the sense that the serialization order is determined within the transactional boundaries but updates are propagated only after the commit of the transaction. This approach does not restrict replica placement but is still primary copy and forbids transactions to access data items with primary copies on different sites.

2.3 ... and eager solutions

Parallel to this work, several suggestions [AAES97, PGS97] have been made to implement eager replication using *group communication systems* such as Transis, Totem or Horus [ea96] and initial efforts have been made to optimize the integration of transaction processing and communication management [KPAS99]. In some cases [AAES97, HAA99], the protocols are quite simplistic and suffer from high abort rates. In other cases, the protocols can be quite difficult to implement in a real database [PGS97]. In all cases, the work is simulation based and little effort has been made to tackle the practical aspects of a real implementation. To address these limitations, we have proposed a suite of replication protocols [KA98, KA] where different degrees of isolation are combined with different message delivery guarantees in order to provide a more complete solution that takes into account abort rates, failures and how databases relax consistency. The results presented in this paper are based on this work. In what follows, we discuss how these ideas can be implemented in a database management system and show that the performance reached favorably compares with that of traditional protocols.

3 Replication Model

Our approach is based on a number of techniques and optimizations which we briefly present in this section. For more details see [KA].

3.1 Reducing message and synchronization overhead

Traditional eager replication protocols [BHG87] coordinate copies one operation at a time. In a system with n nodes and where each transaction consists of m operations, a throughput of k transactions per second requires $k \cdot m \cdot n$ messages per second. Such an approach can never scale. One way to avoid this problem is to bundle writes into a single *write set* message [AAES97, ABKW98]. In lazy replication this is somewhat easier since updates are propagated after the transaction commits. One novel aspect of our approach is to apply this technique in eager replication. We use *shadow copies* [BHG87] to perform updates: write operations are executed on private copies in order to check consistency constraints, capture write-read dependencies and fire triggers. These changes to the shadow copies are propagated to the other sites at commit time, thereby greatly reducing the message overhead and the conflict profile of transactions.

3.2 Localizing read operations

Early replication protocols like read-one/write-all [BHG87] already recognized the importance of keeping read operations local. This implies that read operations are executed only at one site and that no information about them is exchanged among the sites. As a result, read operations have no message costs and no overhead at remote sites, and queries can be kept completely local. This is very desirable but it introduces some complexity regarding read/write conflicts since reads are only seen at the local site.

3.3 Pre-ordering transactions

We use a group communication primitive providing *total order* semantics to multicast the write set and to determine the serialization order of the transactions. The total order guarantees that all sites receive the write sets in exactly the same order. Note that a site sends a message also to itself in order to be able to determine the final total order of a transaction. Each transaction manager uses this order to acquire locks. It requests all write locks for a transaction in a single atomic operation, and then proceeds with the execution of the transaction. By granting the locks in the order in which the transactions arrive, it is guaranteed that all sites perform conflicting updates in the same order. Additionally, transactions never get into a deadlock. Note that this does not imply serial execution since non-conflicting transactions are executed in parallel. Only the access to the lock table is serial. With this approach, we also avoid that transaction response time is determined by the slowest machine. The local site can commit a transaction whenever the global serialization order has been determined and does not wait for the other sites to have executed the transaction. Instead it relies on the fact that the other sites will serialize the transaction in the same way.
Group communication primitives provide a variety of execution semantics. These semantics can be used to optimize the protocols as long as the recovery mechanisms are properly adjusted. In this paper we assume *reliable delivery*, which guarantees consistency on all non-faulty sites [KA].

3.4 An eager replication protocol

The replication protocol we use in this paper executes a transaction in four phases:

I. *Local Read Phase:* Perform all read operations locally. Execute write operations on shadow copies. Acquire the appropriate lock before executing the operation.

II. *Send Phase:* If T_i is read-only, then commit. Else bundle all writes into write set WS_i and multicast it to all sites including the sending site (same delivery order at all sites).

III. *Lock Phase:* Upon delivery of WS_i, request all locks for WS_i in an atomic step:
   1. For each operation $w_i(X)$ on item X in WS_i:
      a. Perform a conflict test: if a local transaction T_j has a granted lock on X and T_j is still in its read or send phase, abort T_j. If T_j is in its send phase, then multicast the decision message *abort* (decision messages are not ordered).
      b. If there is no lock on X, grant the lock to T_i. Otherwise enqueue the lock request directly after all locks from transactions that are beyond their lock phase.
   2. If T_i is a local transaction, multicast the decision message commit (no order requirement).

IV. *Write Phase:* Whenever a write lock is granted apply the corresponding update. A local transaction can commit and release all locks once all updates have been applied to the database. A remote transaction must wait until the decision message arrives and terminates accordingly.

In this protocol, the total order is used to serialize write/write conflicts at all sites. The scheduler has to guarantee that waiting locks are granted in the order in which they appear in the lock queue. Read/write conflicts are also detected during the lock phase (III.1.a). Since read operations are only seen at the local site, we use a straightforward solution and abort local readers when a conflicting writer arrives. This avoids deadlocks and inconsistent executions. The abort is only necessary when the reading transaction is in its read or send phase. In later phases the transaction cannot be involved in a deadlock (see [KA] for details). When a transaction is aborted during the read phase, it is still completely local and no message needs to be sent. When a transaction is aborted during its send phase, the local site must inform the other sites via an abort message. Thus, this protocol requires that the local site sends two messages per transaction, one with the write set and another to confirm that the transaction will commit or abort (this is not a 2PC). The decision messages do not require any ordering semantics. They may be delivered in any order at the different sites and might even arrive before the corresponding write set. Obviously, to abort readers when a writer arrives is problematic. There exist several alternatives [KA], specially using different degrees of isolation like cursor stability or snapshot isolation. For simplicity, we only analyze the presented protocol in this paper.

4 Postgres-R Architecture

Postgres-R has been implemented as an extension to PostgreSQL [Pos98], version 6.4.2, a single node database that supports an extended subset of SQL and uses 2-phase-locking for concurrency control with re-

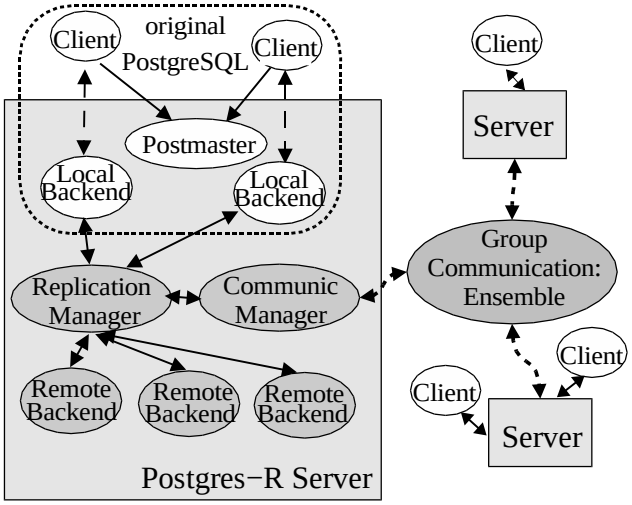

Figure 1: Architecture of Postgres-R

lation level locking. With respect to group communication, we use Ensemble [Hay98], the follow up system to Horus [ea96]. The functionality available in PostgreSQL is still available in Postgres-R but now with replication as an additional feature. All the interfaces provided by PostgreSQL are also available in Postgres-R: embedded SQL, ODBC, etc. The replication features of Postgres-R are usable through SQL, both for data manipulation and data definition. For simplicity, in what follows we assume full replication. In section 6, we discuss how to implement partial replication.

4.1 Basic modules

As shown in Figure 1, a replicated database comprises several nodes (servers), each one of them running an instance of Postgres-R. In the figure, all clear shapes represent original PostgreSQL modules, the shadowed shapes are Postgres-R specific. PostgreSQL is *process-based*. When *clients* want to access the database they send a request to a listener process, called *postmaster*. For each client, the postmaster creates a *backend* process and from then on communication takes place between backend and client. Clients can submit an arbitrary number of transactions (one at a time) until they disconnect. PostgreSQL allows to limit the maximum number of parallel backends. When a given threshold is reached no new clients are admitted.

To implement replication, additional modules are needed to take care of the communication and to deal with remote transactions. In Postgres-R, clients may connect to any server. The transactions of a client are called *local* at the server where the client connects. For each client, the postmaster creates a *local backend* process. To handle *remote* transactions, each Postgres-R server keeps a pool of *remote backend* processes. This pool is created at system startup (with an adjustable startup pool size). When a message of a remote transaction arrives, it is handed over to one of these remote backends. When all existing remote backends are busy and their number has not reached a given threshold (maximum pool size), a new remote backend is started, otherwise the transaction must wait.

Control of the replication protocol takes place at the *replication manager*. The replication manager is a message handling process. It receives messages from the backends (local and remote) and forwards them to the other sites. It also receives the messages delivered by the communication system and forwards them to the corresponding backends.

4.2 Execution of transactions

For local transactions, as long as they are in their *read phase*, they remain within their corresponding local backend. Once the local backend finishes the execution (over shadow copies), it sends the write set to the replication manager and the transaction enters its *send phase*. The replication manager then broadcasts the write set to all sites. When a write set arrives at a site, its replication manager checks whether the write set corresponds to a local or to a remote transaction. This is done using the host name and a transaction identifier included in the write set message. For write sets of local transactions, the replication manager notifies the corresponding local backend and the transaction enters its *lock phase*. In order to perform the lock phase atomically, the local backend acquires a latch on the lock table and keeps it until all locks are enqueued in the lock table. Additionally, the replication manager stops accepting write sets from the communication system until the backend sends a confirmation that all necessary locks have been requested. This guarantees that the lock phases of concurrent transactions are executed in the same order in which the write sets have been delivered. When the replication manager receives the confirmation from the backend that the locks have been requested, it broadcast a commit message. The transaction then enters its *write phase* and whenever a lock is granted the shadow copies become the valid versions of the items.

Up to the time point at which a local transaction T has acquired the latch on the lock table to start the lock phase, it can be aborted due to a read/write conflict. This happens when another transaction T' tries to set a write lock (during its lock phase) and finds a read lock from T. In this case, T' sets an *abort flag* for T. Local transactions check their abort flags regularly during their read and send phases and abort if they are set. If a transaction is in its send phase, it sends an abort message to the replication manager which broadcasts it to the other sites.

For remote transactions, write and decision messages might arrive in any order. If the write set arrives first, the replication manager passes it to an idle remote backend and proceeds like with local write sets. The remote backend will acquire the locks, confirm this to the replication manager, and apply the updates. However, it will wait to terminate the transaction until the replication manager receives the decision message from the local site and forwards it to the remote backend. Once a remote backend finishes executing a transaction, it sends a *ready*-notification to the replication manager, which will add it to the pool of available remote backends. If the decision message arrives first, the replication manager registers this fact and simply proceeds accordingly when the write set arrives.
To implement these procedures, the PostgreSQL *lock table* had to be modified. Usually, a transaction requests a lock and performs an operation before it requests the next lock. In between the two lock requests, other transactions can also acquire locks. In Postgres-R, it is possible to request all write locks in a single step. As a consequence a transaction can have more than one lock waiting and more than one lock granted without the corresponding operation being executed. Additionally, the *backend coordination* of PostgreSQL needed to be adjusted, but the actual data manipulation and commit actions could be reused.

4.3 Shadow copies

In Postgres-R, updates are executed on a shadow copy during the read phase. This is crucial for several reasons. First, a transaction is able to read what it has previously written by reading the shadow copies. Second, constraints can be checked to assure that the write operation is indeed possible. And finally, triggers can be fired that possibly generate further updates (which are then also performed on shadow copies within the scope of the transaction).
PostgreSQL supports shadow copies quite well since it is a *tuple-based multiversion system*. In PostgreSQL each update invalidates the current physical version of a tuple and creates a new version. To determine the valid version, each tuple has two additional fields which contain the identifiers of the creating and the invalidating transaction. A version is visible to a transaction T_i if T_i itself has created it or the creating transaction T_j has already committed. Furthermore, for the tuple to be visible, the field for the invalidating transaction must be empty or the invalidating transaction is either still running or aborted. Thus, a transaction sees its own updates but not the updates of concurrent transactions. Updates trigger the creation of new entries in all relevant indices. To control the table size, PostgreSQL provides a special garbage collector to physically delete all invisible tuples. Thus, the integration of the shadow copy approach into PostgreSQL has been rather straightforward. It should be equally feasible in any multiversion database (e.g., Oracle).

4.4 Locking

PostgreSQL uses logical locking at the relation level. For efficiency reasons, however, it is desirable to have tuple level locking. Thus, we have implemented a simple tuple level locking scheme based on key values. Using shadow copies greatly helps to accomplish this. During the read phase, a local site actually executes the transaction and, therefore, can determine the primary key values of all items that have been modified. Including these key values in the write set allows for logical tuple level locking during the lock phase.
Although shadow copies are not visible until commit time, they require a sophisticated handling of locks to avoid `update/update`, `delete/update` and `insert/insert` conflicts. Assume write operations on shadow copies would not acquire locks and there are two transactions T_1 and T_2:

```
T1:update ATABLE set A1=A1+1 where A-ID=5
T2:update ATABLE set A1=A1+2 where A-ID=5
```

Both might be on the same site or on different sites and they perform the updates concurrently on shadow copies. Now assume, both send their write sets and T_1's write set is delivered before T_2's write set. Since neither T_1 nor T_2 have locks set on the data during the read phase, first T_1's and then T_2's updates will be applied. This results in a non-serializable execution. The problem here is that both operations contain implicit reads. For delete/update conflicts, the problem is incompatible writes. Assume T_1 deleting a tuple and T_2 concurrently updating the tuple and T_1's write set is delivered before T_2's write set. While T_2 could locally update the tuple during the read phase the write phase will fail because T_1 has deleted the tuple. A similar problem arises with two concurrent inserts.
To avoid these problems, we use a similar approach as the multiversion 2-phase-locking scheme proposed in [BHG87]. The approach is also related to *update mode* locks [GR93]. The idea is to obtain a *read-intention-write* (RIW) for all write operations during the read phase. A RIW lock conflicts with other RIW locks and with write locks but not with read locks. As a result, a transaction can perform a write operation on a shadow copy while concurrent transactions can still read the (old) version of the tuple. By using this mechanism, the problems described above are either avoided or are made visible. Conflicts between two local transactions are handled by allowing at most one RIW lock on a data item. Conflicts between local and remote transactions are detected during the lock phase of the remote transactions. In this case RIW locks behave like read locks. If a transaction in its lock phase

wants to set a write lock on a data item, it will abort all local transactions in their read or send phases with conflicting read or RIW locks.
The only problem with RIW locks is that they reintroduce deadlocks. Assume transaction T_1 updates data item x and T_2 updates data item y both holding RIW. If now T_1 wants to set a RIW lock on y and T_2 wants to set a RIW lock on x, a deadlock will ensue. However, such a deadlock only occurs among local transactions in their read phases and therefore can be handled locally. Note that, once a transaction is in its send phase, it will not be involved in a deadlock anymore because write locks have precedence over any other type of locks and conflicting transactions will be aborted.

4.5 Index locking

Locks on index structures need further consideration. Most of these locks are usually short locks not following 2-phase-locking and hence, they can be acquired at any time even during the write phase of a transaction. In B-trees, for instance, while searching for an entry to be updated, PostgreSQL searches along a path in the B-tree, locking and unlocking (short read locks) individual pages until the entry is found. When the entry is found, the short read lock is upgraded to a write lock. Two transactions can follow this procedure at the same time and deadlock when they try to upgrade the lock. In PostgreSQL, such deadlocks occur frequently because each update operation creates a new entry in the primary key index. These deadlocks would not be a problem if they involved only local transactions in their read phase. However, since indices are also used during the write phase, remote transactions could also be involved in such deadlocks. To avoid it, Postgres-R immediately acquires write locks on index pages in the case of update operations.

4.6 The write set

Creating, sending and processing the write set plays a crucial role in our protocols and can have a severe impact on performance. In Postgres-R, we have implemented two alternatives to send a write operation. Either the *SQL statement* is sent or the primary key values of the updated tuples along with the new *physical values* of those attributes that have been modified. In the former case, messages are small but remote sites have more work to do since they need to parse the SQL statement and execute the entire operation. In the latter case, remote sites can be very fast installing updates (specific tuples are accessed via the primary key index), but messages can become quite large.
We have evaluated the performance differences between the two alternatives in terms of message size and execution time by running two tests. The results are shown in Table 1. For comparison reasons the table also shows the execution time in a non-replicated system. We have run two tests. In the first test, a write set contains a single operation updating one tuple. The index on the primary key can be used to find the tuple. In the second test, there is one operation updating 50 tuples. This statement performs a table scan. In both cases, two tuple attributes are modified. Regarding message size, in the 1-tuple case, there are no significant differences between sending statements or the physical updates. However, with 50 tuples, the message with physical updates is quite big and might lead to severe latency and buffer problems in the communication system. Regarding execution time, if the SQL statement is sent or if only one tuple is updated the overhead at the local site is not visible and execution takes as long as in the non-replicated case. But even if the local site must include the physical updates of 50 tuples, the overhead is not very high. The most visible difference, however, is how much faster a remote site can apply the physical updates in comparison to executing the SQL statement. Since the overhead at the local site occurs only once while there are many remote sites, we prefer sending the physical updates as long as message size is not the limiting factor.

	1 tuple		50 tuples	
	SQL	phys.	SQL	phys.
Message Size (Byte)	123	105	125	3634
Execution Time (ms)				
Not Replicated	7		125	
Local	7	7	125	140
Remote	7	1	125	40

Table 1: The Write Set

5 Performance Analysis

5.1 General configuration

PostgreSQL uses a *force* strategy to avoid redo recovery, flushing all dirty buffer pages at the end of each transaction. With this strategy, response times are very poor. This makes it difficult to compare with commercial systems which only flush redo logs to disk. To allow us to use a more "realistic" setting we used the no-flush option offered by PostgreSQL. With this option nothing is forced to disk, not even a log record. This, of course, violates the ACID properties, however the measured response time was better comparable to standard database systems. In future versions of Postgres-R we will correct this limitation.
We have performed 4 experiments. Except for the first experiment, we used a cluster of 15 Unix workstations (SUN Ultra 10, 333 MHz UltraSPARC-IIi CPU, 2MB cache, 256 MB main memory, 9GB IDE disk, switched

Parameters	Ex. 1	Ex. 2	Ex. 3	Ex. 4
Database Size	10 tables of 10000 tuples each			
Tuple Size	appr. 100 Bytes			
# of Servers	1-5	1-15	1-15	1-15
% of Upd. Txn.	100%	100%	100%	varying
# Op. in Upd. Txn.	5	10	1	10
# Op. in Query	-	-	-	1 scan
# of Clients	5	20	20	3 p. serv.
Submission rate in tps in the entire system	10	20-50	40-200	15-225

Table 2: Parameters settings

full-duplex Fast Ethernet). We did not have exclusive access to the cluster. For all our experiments, we use the physical copy approach in which servers only apply the physical updates of remote transactions.

In all our experiments, the database consists of 10 tables each containing 1000 tuples. We did not consider larger databases since this will only reduce the conflict profile. Each table has the same schema: two integers (one being the primary key `t-id`, the other denoted below as `attr1`), one 50-character string (`attr2`), one float (`attr3`) and one date (`attr4`) attribute. For each table there exists one index for the primary key.

Update transactions have operations of the type

```
update t-name set attr1='x', attr2=attr2+4
where t-id=y
```

where x is randomly chosen text and y is a randomly chosen number between 1 and 1000. The relevant tuple is found by searching the index on the primary key.

Transactions are submitted by clients which are evenly distributed among the servers. The interarrival time between two submissions is exponentially distributed. The submission rate (also referred to as workload) is determined by the number of clients and the mean interarrival rate for each client. The system throughput is equal to the submission rate unless the system is saturated. Whenever a transaction is aborted, the client resubmits it immediately.

Table 2 summarizes the parameters of all experiments. As performance indicator, we analyze the response time of local transactions, i.e., the time from which the client starts a transaction until the client receives the commit. For comparison: in a single user, single node system, an update transaction with 10 operations takes around 75 ms, with 1 operation 9 ms.

5.2 Experiment 1: distributed 2 phase locking

In a first experiment we compared standard distributed locking with Postgres-R. To do so we use a commercially available implementation of eager replication based on standard distributed locking. The experiment was conducted with 5 instances of a

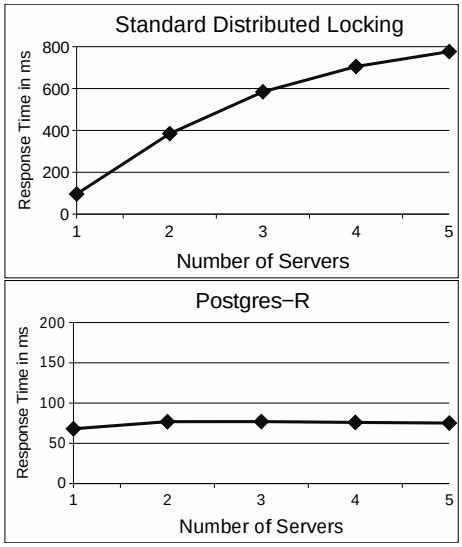

Figure 2: Comparison of distributed locking and Postgres-R

database product installed on PCs (266MHz, 128MB main memory, two local disk [4GB IDE, 4GB SCSI], switched full-duples 100Mbit Fast Ethernet). The workload consists of only update transactions, with 5 operations each. The number of clients was fixed to 5 and the interarrival rate per client to 500 ms for a total submission rate of 10 tps. The number of replicated nodes was varied from 1 (no replication) to 5.

The results are shown in Figure 2. As predicted by Gray et al., we observe a clear degradation in the distributed locking solution as the number of servers increases. In addition to longer response times, we also observe significantly higher abort rates and decreasing throughput beyond 2 nodes.

In comparison, the performance of Postgres-R proved to be stable. Please note, that this comparison can only be relative since different hardware platforms are used and the underlying database systems differ considerably. Still, the test demonstrates that – at least for this relatively small load (10 tps) – the dangers of replication seem to have been circumvented in Postgres-R. To find out whether this is indeed the case, we performed three other experiments varying the workload and communication overhead as well as testing the scalability of Postgres-R.

5.3 Experiment 2: workload

Conventional eager replication protocols do not cope very well with increasing loads. The next step is to analyze the behavior of Postgres-R for high update workloads. To do so, we run tests with different workloads and system sizes from 5 to 15 nodes. For all runs, transactions consist of 10 update operations and

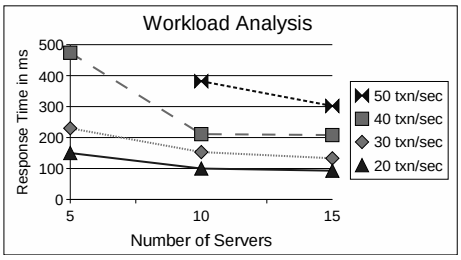

Figure 3: Response time of Postgres-R for different workloads and number of nodes

there are 20 clients in the system.

As the results in Figure 3 show, for small loads we could run the tests with 5, 10 and 15 nodes. For higher loads, the 5 node configuration was saturated. The results clearly demonstrate that the response times improve as we increase the number of replicated nodes due to the increased processing capacity. This is exactly the behavior that is needed to improve performance by using replication.

To understand why Postgres-R can take advantage of the increased processing capacity, we need to look at how transactions are being processed. Regarding remote transactions, nodes only install the changes without having to execute the SQL statements (see Table 1). As a result, remote transactions use significantly less CPU than local ones which allows to run additional transactions or reduce overall response times. In addition, since updates on remote sites are so fast and applied in a single step, the corresponding locks are held for a very short time. With this, conflict rates are very low (in the tests abort rates never exceeded 5%).

Another important point is that the coordination overhead seems to have very little impact. First, we observed a generally small communication overhead (message delays were between 5 and 10 ms and constructing the write set added only a few milliseconds). Second, the local node does not need to wait until the remote sites have executed the transactions but only waits until the write set is delivered.

We can conclude that the techniques implemented in Postgres-R can be used to increase the processing capacity of a database by using replication. We only tested up to 15 nodes due to the practical limitations of setting up the experiment. With the results obtained we are confident that Postgres-R can cope with both higher loads and more nodes.

5.4 Experiment 3: communication overhead

One of the problems of using group communication systems is the poor performance that many of them exhibit. The claim that Postgres-R can tolerate more

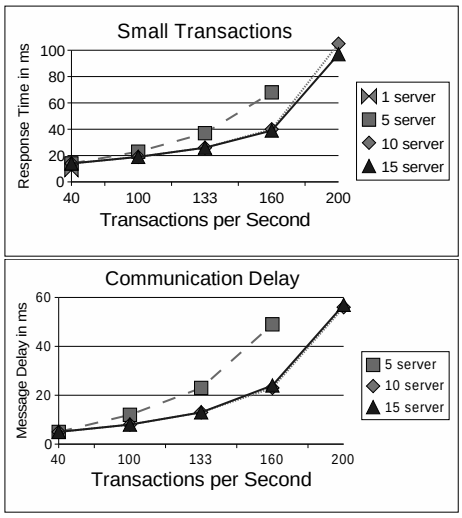

Figure 4: Response time and communication delay for small transactions

than 15 replicated nodes is conditional to proving that the communication system used actually scales up. In this third experiment, we analyze how the communication system handles high message rates. The goal is to test whether high transaction loads can collapse the communication system and whether message delays can severely affect response times.

In the previous experiments, the number of messages never exceeded 100 messages per second. Up to then, the communication system was not the bottleneck. In order to stress test the system, we performed an experiment with very many, very short transactions. These transactions consist of only one operation, thus, the write set is small but the communication overhead has a bigger impact on the overall response time. Again, we use 20 concurrent clients generating a throughput between 40 and 200 transactions per second.

Response times and message delay are shown in Figure 4. Clearly, as the number of messages in the system increases, the communication system becomes slower. Transaction response times vary proportionally to the message delay as the similarity between the slopes in the figures indicate. A resource analysis has shown that the communication process requires the most CPU at high transaction loads. This means, that the message delay is due to increased message processing requirements (for message buffering, determining the total order etc.) and not to a shortage of network bandwidth. Observe, however, that the number of nodes plays no role on the communication congestion. It is only the submission rate that has an effect. Thus, replication can still be used to improve performance. A 1-node system, while slightly faster at 40 tps, cannot cope with 20 clients and a workload of

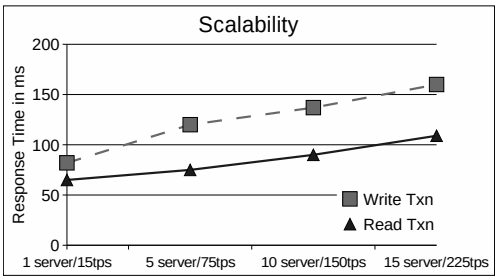

Figure 5: Response time for update transactions and queries

100 tps due to process management and log and data contention while the replicated systems do.
As a summary of this experiment, communication overhead is a factor to take into consideration but it only started to play a limiting role under high update transaction loads (over 150 tps). Then, an efficient communication module is crucial. We believe, however, that when read transactions are considered (which do not require communication), the mechanisms of Postgres-R can provide good performance over an even wider range of loads.

5.5 Experiment 4: scalability

This last experiment tests the scalability of Postgres-R using a more realistic workload of update and read-only transactions (queries). Update transactions have 10 update operations and queries are of the form
`select avg(attr3), sum(attr3) from t-name`
scanning an entire table. There are three clients per site, one submitting an update transaction each 1 second, the two other submitting queries each 150 ms. Thus, the load per node is around 15 tps with a 14 to 1 rate between read and update transactions.
The response times for both read and write transactions are shown in Figure 5. As pointed out above, by considering queries, we are able to achieve higher throughput (up to 225 tps in a 15-node system). The response times increase with the number of nodes but are reasonable if we take into account that the absolute number of update transactions (that must be applied everywhere and create conflicts), increases constantly. In fact, conflicts start to become a problem at higher loads. The way to address this limitation is to use alternative isolation levels. In our specific setting, since the queries only set a single relation level lock they cannot be involved in any deadlock, and hence, we do not abort them. Still, queries and update transactions delay each other. As an alternative, a hybrid protocol could combine serializability for update transactions and provide a snapshot for queries [KA]. In that way, updating transactions never conflict with queries and are not delayed by them. Such a hybrid protocol practically eliminates conflicts at most loads and will allow to scale Postgres-R even further.

6 Crash Recovery, Administration and Partial Replication

Postgres-R has been designed as a system able to cope with issues like failures and partial replication which are often ignored in research. We have also implemented the administrative tools necessary to set up and maintain the system.

6.1 System configuration

One of the main problems of replication is how to dynamically change the system without having to stop processing. We have designed Postgres-R to work in cluster environments where failures and configuration changes can occur quite frequently. We support this by using the group communication services. As nodes leave (because of failures or shutdowns) or join (new or recovering nodes), the group communication module creates different *views* in the computation. Every time there is a change in the number of nodes, the communication system switches to a new view and informs the replication managers via a view change message.
In the case of failures, when a working sites receives the corresponding view change message, it can identify the active transactions originating at the failed site. In [KA] we show that active transactions from failed nodes can be safely aborted without compromising consistency at the non-faulty nodes.
Upon recovery, or when a new node is added to the system (also triggering a new view), a peer node has to provide a copy of the current database. The transfered data must contain the updates of all write sets that were delivered in the old view (without the joining node). PostgreSQL provides a feature which extracts the database schema and all tuples from a given database to transfer it to another database. We use this feature to install the database in the new node. While the data transfer takes place other nodes in the system can continue processing transactions. The new node will receive these messages (since they execute in the new view) but delay their execution until all data is installed and only then apply the updates. Once all this is done, the new node will allow clients to connect and proceed from then on like a normal node.

6.2 Partial replication

For simplicity in the exposition, we have assumed full replication (all data is replicated at all nodes). Partial replication, however, is an important issue that needs to be addressed. Partial replication means each data item can have one or more copies residing on arbitrary

sites. With this, data propagation and enforcing serializability can become quite complex. To tackle this problem Postgres-R implements a *client makes it right* approach where the local node sends all updates of a transaction to all sites, regardless of who has a copy. The nodes receiving this information have to identify which updates need to be done locally and which ones can be ignored. With this strategy, we still can send all updates in a single message and the total order can be used to determine the serialization order following a protocol identical to the one discussed in the paper. The overhead involved is not as high as it may seem. If changes are propagated as physical updates, checking whether a tuple is local or not can be done very quickly. The advantage is that a site does not need to know where the particular copies of a data item reside in order to send the write set. Furthermore, subscribing or unsubscribing to a a data item can be handled with little administrative overhead.

7 Conclusion

Database replication is an increasingly important topic. New computing environments will demand innovative solutions and flexible mechanisms that can support different forms of replication. In particular, eager replication is extremely useful in cluster based systems. Unfortunately, existing commercial products tend to support mainly lazy replication. Similarly, the research focus has shifted towards lazy approaches and this is likely to prevent that future products support eager replication.

In this paper, we prove that eager replication is feasible using the adequate techniques. We have proposed a simple replication protocol, showed how it can be incorporated into a real database management system and analyzed its performance. That is, we have worked out the engineering issues that a protocol needs to address to actually work in practice, issues that have been largely ignored in the literature.

As part of future work, we are developing more sophisticated management tools for Postgres-R and extending the range of replication possibilities by implementing a wide range of eager and lazy protocols.

References

[AAES97] D. Agrawal, G. Alonso, A. El Abbadi, and I. Stanoi. Exploiting atomic broadcast in replicated databases. In *Proc. of Euro-Par*, 1997.

[ABKW98] T. A. Anderson, Y. Breitbart, H. F. Korth, and A. Wool. Replication, consistency, and practicality: Are these mutually exclusive? In *Proc. of the SIGMOD Conf.*, 1998.

[AES97] D. Agrawal, A. El Abbadi, and R. C. Steinke. Epidemic algorithms in replicated databases. In *Proc. of PODS*, 1997.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison Wesley, Massachusetts, 1987.

[BKR+99] Y. Breitbart, R. Komondoor, R. Rastogi, S. Seshadri, and A. Silberschatz. Update propagation protocols for replicated databases. In *Proc. of the SIGMOD Conf.*, 1999.

[CRR96] P. Chundi, D. J. Rosenkrantz, and S. S. Ravi. Deferred updates and data placement in distributed databases. In *Proc. of the Int. Conf. on Data Engineering*, 1996.

[ea96] D. Powel et al. Group communication (special issue). *Communications of the ACM*, 39(4):50–97, 1996.

[GHOS96] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proc. of the SIGMOD Conf.*, 1996.

[GN95] M. Gallersdörder and M. Nicola. Improving performance in replicated databases through relaxed cohrency. In *Proc. of the VLDB Conf.*, Zürich, Switzerland, 1995.

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

[HAA99] J. Holliday, D. Agrawal, and A. El Abbadi. The performance of database replication with group multicast. In *Proc. of the Int. Symp. on Fault-Tolerant Computing (FTCS)*, 1999.

[Hay98] M. Hayden. The ensemble system. Technical report, Departement of Computer Science, Cornell University TR-98-1662, January 1998.

[KA] B. Kemme and G. Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Transactions on Database Systems.* accepted for publication.

[KA98] B. Kemme and G. Alonso. A suite of database replication protocols based on group communication primitives. In *Proc. of the Int. Conf. on Distributed Computing Systems (ICDCS)*, 1998.

[KB91] N. Krishnakumar and A.J. Bernstein. Bounded ignorance in replicated systems. In *Proc. of PODS*, 1991.

[KPAS99] B. Kemme, F. Pedone, G. Alonso, and A. Schiper. Processing transactions over optimistic atomic broadcast protocols. In *Proc. of the Int. Conf. on Distributed Computing Systems (ICDCS)*, 1999.

[PGS97] F. Pedone, R. Guerraoui, and A. Schiper. Transaction reordering in replicated databases. In *Proc. of the Symp. on Reliable Distributed Systems*, 1997.

[PL91] C. Pu and A. Leff. Replica control in distributed systems: An asynchronous approach. In *Proc. of the SIGMOD Conf.*, 1991.

[PMS99] E. Pacitti, P. Minet, and E. Simon. Fast algorithms for maintaining replica consistency in lazy master replicated databases. In *Proc. of the VLDB Conf.*, 1999.

[Pos98] PostgreSQL. *v6.4.2 Released*, January 1998. http://www.postgresql.org.

[SAS+96] J. Sidell, P.M. Aoki, A. Sah, C. Staelin, M. Stonebraker, and A. Yu. Data replication in Mariposa. In *Proc. of the Int. Conf. on Data Engineering*, 1996.

[Sta94] D. Stacey. Replication: DB2, Oracle, or Sybase. *Database Programming & Design*, 7(12), 1994.