Postgres-R(SI): Combining Replica Control with Concurrency Control based on Snapshot Isolation

Shuqing Wu Bettina Kemme
School of Computer Science, McGill University, Montreal
{swu23,kemme}@cs.mcgill.ca

Abstract

Replicating data over a cluster of workstations is a powerful tool to increase performance, and provide fault-tolerance for demanding database applications. The big challenge in such systems is to combine replica control (keeping the copies consistent) with concurrency control. Most of the research so far has focused on providing the traditional correctness criteria serializability. However, more and more database systems, e.g., Oracle and PostgreSQL, use multi-version concurrency control providing the isolation level snapshot isolation. In this paper, we present Postgres-R(SI), an extension of PostgreSQL offering transparent replication. Our replication tool is designed to work smoothly with PostgreSQL's concurrency control providing snapshot isolation for the entire replicated system. We present a detailed description of the replica control algorithm, and how it is combined with PostgreSQL's concurrency control component. Furthermore, we discuss some challenges we encountered when implementing the protocol. Our performance analysis based on the TPC-W benchmark shows that this approach exhibits excellent performance for real-life applications even if they are update intensive.

1. Introduction

For a long time, data replication has been considered an excellent solution to increase throughput (more replicas can serve more requests), decrease response times (distribute the load and access the local replica), and provide fault-tolerance. Replication, however, has the challenge of replica control. Changes submitted to one replica have to be applied at the other replicas such that the different copies of the database remain consistent despite concurrent updates. Standard correctness criteria is 1-copy-serializability, i.e., the concurrent execution of a set of transactions on the different replicas should have the same effect as a serial execution on a centralized database. As such, replica control has to be combined with or at least must be aware of the concurrency control mechanism that determines the serialization order at each individual replica. Early research solutions focused on fault-tolerance [5], and were seldomly implemented in commercial systems, which mostly offered ad-hoc solutions violating traditional transactional properties in order to achieve acceptable performance [15]. A thorough analysis by Gray et.al. in 1996 [16] claimed that existing approaches scale badly, and are not suitable for modern applications. Their analysis revived research in database replication leading to many new solutions that attempt to eliminate the limitations pointed out by [16], while still providing global serializability, e.g., [3, 27, 7, 18, 21, 6, 22, 26, 25, 30, 1, 19, 8, 23, 14, 2].

For instance, Postgres-R [21] implements a locking based replica control algorithm by extending PostgreSQL, version 6. It uses special multicast primitives provided by group communication systems to propagate write operations. These primitives order concurrent messages and deliver them to all replicas in the same order. All replicas execute conflicting operations according to this order guaranteeing global serializability.

While many of the other proposals were analyzed using analytical or simulation based studies, very few were implemented as prototypes or real systems. Of those, most use a middleware layer and keep the database replicas nearly unchanged. As such, the middleware has to implement its own concurrency control component, and faces the challenge that only the SQL statements are visible but not the individual tuples to be accessed. In contrast, Postgres-R [21] integrates replica control into the kernel of a database system. The main advantage of such approach is that it can take advantage of internal components. For instance, it can directly interact with the often tuple-based concurrency control of the database system, and does not need to implement its own concurrency control mechanism. Furthermore, direct access to the tuples and/or logs is given allowing for an efficient propagation of changed tuples. Another advantage is that the replica tool comes within the same software package as the database system making installation and usage easier. Commercial systems are able to sell their replication modules at high price for exactly these reasons.

However, while serializability is the predominant correctness criteria used in research, more and more database systems only offer the isolation level *Snapshot Isolation* (SI), e.g., Borland [11], Oracle [12], and version 7 of PostgreSQL [17]. This isolation level can be implemented using multi-version concurrency control. Snapshot isolation allows some non-serializable executions, but it provides much more concurrency for read-only transactions, and hence is very useful for read intensive applications. The basic idea is to keep several versions of a data object. Read operations read from a committed ”snapshot” of the database, and work completely independent from writes. Write operations of concurrent transactions, however, are disallowed.

In this paper, we present Posgres-R(SI), the newest version of Postgres-R, providing replication for PostgreSQL 7.2. Posgres-R(SI) has a similar architecture as the original Postgres-R using a total order multicast. What is new is the integration of replica control with the multi-version concurrency control algorithm of PostgreSQL, version 7, in order to detect write/write conflicts and determine the snapshot of read operations. This required us to obtain a detailed understanding of PosgreSQL's multi-version system, and made it a challenge to extend PostgreSQL in a modular way without really changing existing components.

In summary, the paper presents a thorough analysis of a concurrency control mechanism providing snapshot isolation, and a replica control algorithm based on this mechanism. Our implementation presents a modular extension of the PostgreSQL database system, providing replication without significant changes to the original system. Replication is transparent to the user providing it with the same isolation level as the centralized system. Furthermore, Postgres-R(SI) provides excellent performance in terms of throughput, scalability, and response times.

The rest of the paper is structured as follows. Section 2 presents related work. Section 3 introduces background information. Section 4 describes the algorithm, and Section 5 presents the implementation. Section 6 provides a performance evaluation. Finally, Section 7 concludes the paper.

2. Related Work

In the last decade, a vast body of research has proposed replica control solutions based on various concurrency control mechanisms, e.g., locking [18, 21, 22, 23, 8], optimistic [27], multi-version [1], and serialization graph [3] based techniques providing serializable execution, or mechanisms that provide lower levels of isolation [21, 29]. These replica control algorithms can be categorized by two parameters as described in [16]. The first parameter describes the time point of coordination among replicas. In lazy schemes, a transactions executes locally and changes are only propagated after the commit providing fast user response time but transactions might be lost if a replica fails between committing and propagating changes. In contrast, eager schemes coordinate before the transaction commits. This can guarantee consistency at all times but increases response times. The second parameter determines where updates can be submitted to. In a primary copy approach, only the primary copy of a data object accepts updates propagating them to the secondary copies. In contrast, using update everywhere, all replicas accept updates and take charge of coordination.

Primary copy approaches allow for a loose coupling between replica and concurrency control, only requiring that secondaries apply conflicting updates in the same order they were executed at the primary. [7, 26, 25, 30, 14, 29] follow this approach putting their emphasis on how to guarantee that read-only transactions on the secondary copies read consistent data. Among them, [30, 29] provide middleware based prototypes. The main problem of primary copy approaches is its restrictive handling of updates. Updates may only be submitted to the primary and transactions that want to update objects with different primaries are disallowed.

Update everywhere is more flexible but requires a more sophisticated integration with concurrency control. If combined with lazy propagation, the problem arises that conflicts of transactions executing on different replicas can only be determined after transaction commit requiring to undo committed transactions. Hence most research focuses on eager solutions. Many proposals use, as we, group communication systems with total order multicast to help serializing transactions, e.g. [27, 18, 21, 22, 25, 23]. Except of Postgres-R, they are only evaluated on a simulation basis [27, 18, 22] or within a middleware [23]. In other approaches, a middleware based scheduler provides concurrency control on a table basis [8, 1]. [23, 2] additionally analyze load-balancing issues in middleware based systems. Most of them require to know all tables accessed by a transaction in advance limiting flexibility.

There has been little work so far to combine replica control with concurrency control based on snapshot isolation. [29] presents a primary-copy based middleware solution which relies on the underlying database system for concurrency control. [31] provides a snapshot isolation algorithm for a federated database which could also be used for replica control. In [22], we propose a replica control solution based on snapshot isolation. Clearly, the work we present here has been inspired by our previous work. However, the concurrency control algorithm of PostgreSQL is very different than the one assumed in [22]. Version labeling and management is different, and PostgreSQL uses locking to detect write/write conflicts while [22] proposes an optimistic scheme. As such, the algorithm within Posgres-R(SI) has little to do with the algorithm proposed in [22].

Most commercial systems provide replication solutions that are integrated into the database kernel, providing various replication schemes. Most of them, however, are lazy, primary copy schemes [28].

3. Background

3.1. Transactions and Isolation Levels

A transaction T_i is a sequence of read $r_i(X)$ and write operations $w_i(X)$ on data objects X. T_i must run atomically, i.e., either all operations succeed and T_i commits (c_i), or none of its operations has effect on the database and it aborts (a_i). We define two transactions to be concurrent if neither terminated (commit/abort) before the other started. Isolation levels describe how the system handles conflicting operations of concurrent transactions. Two operations conflict if they are from different transactions, access the same object and at least one is a write. The strongest isolation level is serializability guaranteeing that the concurrent execution of a set of transactions has the same effect as a serial execution. [4] introduces an isolation level called snapshot isolation (SI) which is used by several multi-version database systems where write operations create new object versions. We denote an object version to commit once the transaction that created the version commits. A concurrency control system providing SI must obey the following rules. (i) Each first $w(X)$ of a transaction T on object X creates a new version of X, (ii) subsequent $r(X)/w(X)$ of T access the newly created version, and (iii) a $r(X)$ of T not preceded by a $w(X)$ of T, reads the last version of X that committed before T started. Finally, if two concurrent transactions write object X, at least one of them must abort.

SI separates read and write operations. Reads, never blocking or being blocked, do not interfere with writes since they read from a committed snapshot. However, SI does not guarantee serializability. Assume T_1 and T_2 both read objects X and Y and then update one of the objects. SI allows the execution $r_1(X), r_2(X), r_1(Y), r_2(Y), w_1(X), w_2(Y)$ since the transactions update different objects. However, it is not serializable since in any serial execution either T_1 would have read the update performed by T_2 or vice versa.

3.2. Group Communication Systems (GCS)

In a group communication system (GCS) [9], a group consists of a set of members. A member can multicast a message (via the GCS) to all group members (including itself) or send point-to-point messages. Messages are guaranteed to be delivered if no failures occur, otherwise at-most-once. Different multicast protocols provide different ordering and reliability properties. In our context, we are interested in *uniform-reliable delivery* (if any member receives a message – even if it fails immediately afterwards – all members receive the message unless they fail) guaranteeing basically all-or-nothing on the message level. Futhemore, we are interested in totally ordered messages (if two members receive messages m and m', they both receive them in the same order). GCS also maintains a view of the currently connected members. Whenever the view configuration changes, the GCS informs all member applications by delivering a view change message. The typical property for group membership is *virtual synchrony*: If members p and q receive both first view V and then V', they receive the same set of messages while members of V. The GCS provides explicit join and leave primitives. Additionally, crashed members are automatically removed from the view using a failure detection mechanism. Since failure detectors cannot be perfect in asynchronous environments, the GCS might exclude a correct member. In this case, we assume that the replica shuts down itself and is considered failed. Current GCS have excellent performance in LANs. For instance, Spread [32] only needs a few miliseconds for a total order uniform-reliable message delivery, and is able to handle hundreds of messages per second of this message type.

3.3. Replica Control based on GCS

In this paper we follow the approach of [22] to exploit GCS for replica control, and we present the principle ideas very shortly. A transaction T_i can be submitted to any replica. This replica is T_i's *local* replica and T_i is local at this replica. All other replicas are *remote* replicas for T_i and T_i is *remote* at these replicas. T_i is first completely executed at the local replica, and write operations are collected within a *writeset*. At commit time, the writeset is multicast to all replicas using the total order multicast. All replicas now use the total order delivery to determine the serialization order. Whenever two operations conflict they must be executed in the order the writesets were delivered. Since this is the same at all replicas, all replicas serialize in the same way. No complex agreement protocol or distributed concurrency control is necessary. The different protocols proposed in [22] use different local concurrency control protocols as their basis. Furthermore, uniform-reliable delivery is used to avoid lost transactions. When the sender receives a writeset itself it knows that everybody else will or has received it. Hence, it is safe to commit/abort a transaction locally because the other replicas will do the same. This model is different than the model assumed by [16] for eager update everywhere replication, and hence, avoids many of the limitations pointed out by [16]. Instead of having messages for each operation, only one message exchange occurs per transaction. Instead of running an expensive 2-phase-commit requiring each replica to execute the transaction before it can commit, it can commit locally without waiting that other replicas have executed the writeset. This leads to decreased response times and conflict rates.

When replicas fail, the GCS informs the remaining replicas. They simply can continue as a smaller group. Recovery requires to transfer the current database state to the recovering replica. We will shortly discuss this in Section 4.3.

4. Replica Control providing SI

In this section, we first present the centralized concurrency control algorithm of PostgreSQL, version 7.2, which we call SI-P. Then, we propose a replica control algorithm SI-PR, which is based on SI-P. For simplicity of description, we only consider SQL update statements. It is easy to extend the algorithms for insert and delete statements, and Postgres-R(SI) supports them.

4.1. SI-P: Concurrency Control in PostgreSQL

In PostgreSQL, each transaction T_i is assigned a unique identifier TID_i when it starts. This identifier will be used for labeling tuple versions and detect conflicts.

Version System In PostgreSQL, each update on tuple X creates a new version of X. We denote a version created by a transaction that committed (aborted) a *committed* (*aborted*) version. An important characteristic is that a write operation $w(X)$ have to acquire an exclusive lock on X which is only released at the end of transaction. As a result, there are never two transactions concurrently creating new versions of the same tuple. With this, we define as *valid* version of X, the version of X created by the last committed transaction that updated X. There is always exactly one valid version of X. Finally, we denote as *active* version of X, a version created by a transaction that is still active. There is at most one active version of X in the system. In Figure 1 at time $t3$, both $V0$ and $V1$ are committed, $V1$ is valid, and $V2$ is active.

Each tuple version V is labeled with two TIDs. t_xmin is the TID of the transaction that created V, and t_xmax is the TID of the transaction that invalidated V due to an update creating a new version. That is, when a transaction T_i performs an update on a tuple X, it takes the valid version V of X, and makes a copy V^c of V. Furthermore, it sets V's t_xmax and V^c's t_xmin to TID_i. In Figure 1 at time $t3$, $T3$ takes the version $V1$ to make a new version $V2$.

Two concurrent transactions may not perform write operations on the same tuple X. Therefore, before a transaction T_i performs its first write operation on tuple X, it performs a version check to see whether there is any concurrent transaction T_j that updated X and already committed. For that, T_i looks at the valid version of X and checks whether t_xmin is the TID_j of a concurrent transaction T_j. If this is the case, a conflict is detected, and T_i will abort. In Figure 1, $T2$ must abort at time $t2$ because of valid version $V1$ with $t\_xmin = T1$ and $T1$ committed after $T2$ started.

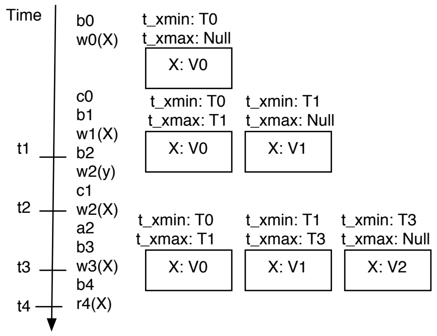

Figure 1. Version creation during execution

When a transaction T_i performs a read operation on X after writing X, it reads its own, newly created version. Otherwise, it reads the version created by transaction T_j such that T_j committed before T_i started and there is no other transaction T_k that updated X and committed after T_j committed and before T_i started. We denote this as T_i's *visible* version of X. Using t_xmin and t_xmax, the visible version can easily be determined. t_xmin must be the TID_j of transaction T_j such that T_j committed before T_i started. t_xmax must be either NULL, refer to an aborted transaction, or refer to a concurrent transaction (invalidation is ignored by T_i independently of whether the concurrent transaction is active or already committed). With this, T_i reads the last committed version as of the time of T_i's start. Note that read operations do not change the t_xmin/t_xmax of any version. In Figure 1 at time $t4$, $T4$ reads $V1$ ($T1$ committed before $T4$ started), and not $V2$.

Determining the snapshot In order to determine valid and visible tuple versions, a transaction must know the status of other transactions. PostgreSQL keeps information about each active transaction T_i in shared memory. Among this information is a `snapshot` struct containing $xmax$, the TID for the next transaction which will start just after T_i, and xip, a list of the $TIDs$ of all active transactions at the moment when T_i starts. We denote as $T_i^{conc} = \{TID \in xip \vee TID \geq xmax\}$ the set of concurrent transactions whose updates are invisible to T_i. All others have already terminated before T_i started. If they committed, their versions might be visible. We denote these transactions as $T_i^{com} = \{TID | TID \notin T_i^{conc}$ and T_{TID} committed$\}$. In order to keep track of the outcome of transactions, whenever a transaction terminates, PostgreSQL inserts a commit/abort log entry into a file called $clog$. The tail of $clog$ is kept in main memory, and $clog$ provides a fast method that takes as input a TID and returns the outcome (commit/abort) of the corresponding transaction. As an example, when $T2$ starts in Figure 1 at $t1$, xip is $\{TID_1\}$, $xmax = TID_3$, and $clog$ contains a commit entry for $T0$.

The SI-P protocol Each transaction T_i has two phases

Execution Phase

- When T_i performs $w_i(X)$ on tuple X for the first time, it first performs a *version check*. It reads the valid version V of X and checks whether $t\_xmin \in T_i^{conc}$. If this is the case, then T_i is aborted (see termination phase). Otherwise ($t\_xmin \in T_i^{com}$), T_i requests an exclusive lock for X. If the lock is granted immediately, T_i makes a copy V^c of V, and sets t_xmax of the old version V and t_xmin of the new version V^c to its own TID_i [1]. If there is already a lock on X, T_i's request is appended to a waiting queue for X. Upon being woken up by the transaction releasing the lock on X, T_i starts all over again with the version check.
- When T_i performs a successive write $w_i(X)$, it simply uses the previously created version V^c.
- When T_i performs $r_i(X)$, it reads its own version V^c if existing, or it reads the visible version V of X, i.e., $t\_xmin \in T_i^{com}$ and $t\_xmax \notin T_i^{com}$ [2].

Termination Phase. Upon the commit request or abort for T_i, T_i updates $clog$, releases all locks, and wakes up all transactions waiting for one of these locks.

Note that the version check happens before requesting the lock and will be repeated if the transaction has to wait for the lock. When a transaction T_i holding the lock commits and wakes up a waiting transaction T_j, T_j's version check will fail since now T_i's version is valid and T_i is concurrent to T_j. If T_i aborts, T_j's check will succeed, and it will again request the lock. If several transactions are waiting, all are woken up and perform the check, and either abort, or compete again for the lock. For example, in Figure 1, assume that $T2$ submits $w2(X)$ at the time the figure shows $w2(y)$. At this time, $T1$ holds a lock on X. When $T1$ releases the lock at $c1$, $T2$ repeats and fails the check.

4.2. SI-PR: Replica Control based on SI-P

Global Transaction Identifiers (GID) $TIDs$ are local at each replica and will differ at the different replicas (which all have different read-only transactions). Many components of PostgreSQL use $TIDs$ in different ways, and hence, we do not want to change their generation. Therefore, each transaction receives a local TID at start as before. In order to compare transactions across replicas, each update transaction additionally receives a global identifier GID, which will be the same at all replicas. The GID of a transaction will be used to match the different local TIDs created on different replicas. We generate GIDs without extra coordination overhead by using the total order in which writesets are delivered. We keep a GID counter at each replica. Whenever a writeset is delivered, the counter is increased and its current value assigned as GID to the corresponding transaction. Furthermore, each replica keeps an internal table that allows for a fast matching between TID and corresponding GID.

The protocol In SI-PR, we have to distinguish between local and remote transactions. As in SI-P, local transactions perform operations step by step whenever a statement is submitted. Remote transactions, however, only have write operations that are all known at the time of writeset delivery. Furthermore, we must guarantee that conflicting operations of both local and remote transactions are executed in the order of writeset delivery. In order to achieve this without too much complexity, we first present an algorithm that executes all remote transactions serially. More precisely, whenever a writeset is delivered for either local or remote transaction, the transaction has to completely terminate before the next writeset is delivered. We indicate this as *atomic* in the algorithm. At the end of this section, we extend the algorithm to allow execution of non-conflicting writesets.

Local Transaction

Execution Phase: The execution phase of a local transaction T_i is the same as in SI–P with some additions. For each $w_i(X)$ that passes the version check (on $TIDs$) and receives the lock, T_i takes the valid and visible version V created by T_j (t_xmin=TID_j), makes its own copy V^c of V and performs the update on V^c. At the same time, it retrieves the GID_j of T_j from the internal table. Both the new version V^c of X and GID_j are added to the writeset.

Send Phase: When T_i submits the commit request, and T_i is read-only, it commits immediately. Otherwise the writeset WS_i is multicast to all replicas using total order multicast.

Commit Phase (atomic): Upon delivery of the writeset of T_i, its GID_i is generated. If T_i was not yet aborted, GID_i is added to the internal table together with TID_i, $clog$ is updated (commit entry), and all locks of T_i are released waking up all transactions waiting for one of these locks.

Abort Phase: Upon abort, $clog$ is updated (abort entry), and all locks are released. If the first waiting transaction for a lock is a remote transaction, only the remote transaction is woken up. Otherwise, all waiting transactions are woken up. (Explanation see below).

Remote Transaction (atomic): Upon delivery of a remote writeset WS_i, a transaction T_i is started, its GID_i generated, and (GID_i/TID_i) added to the internal table.

Version Check and Early Execution: For each tuple version V^c of X in WS_i, the following actions are performed.

[1] Since V is valid, T_i passed the check and has a lock, t_xmax of V is either $NULL$ or the TID of an aborted transaction.

[2] Here, t_xmax might contain the TID_j of concurrent transaction T_j, but V is still visible to T_i because T_i ignores T_j's invalidation of V.

1. T_i retrieves the local valid version V of X. Instead of performing the version check according to local $TIDs$, it performs it using the $GIDs$. That is, it retrieves from the internal table the GID_k belonging to t_xmin of V, and compares it to the GID_j attached to V^c in WS_i.
2. If the GIDs are different, a transaction had updated X and committed since T_i executed on its local replica. Hence, the check fails, and T_i aborts.
3. If the GIDs are the same, the valid version of X is still the same as when T_i executed at its local replica. The version check succeeds, and T_i requests a lock for X.
4. If the lock is granted immediately, T_i sets t_xmax of V and t_xmin of V^c to its own TID_i.
5. If there is already a lock on X, this lock belongs to an active local transaction T_k which either has not yet sent its writeset or whose writeset has not yet been delivered (no other remote transaction is active; and local transactions whose writesets had been delivered are terminated). T_k should be aborted because T_i's writeset is delivered first, and hence, T_i should be serialized before T_k. However, we do not abort immediately but wait until T_i has passed the checks on ALL tuples in WS_i.

Late Execution: If T_i has not yet aborted, it has passed the version checks for all tuples, and performed some of the updates. At this time point, we know that T_i will commit. It now has to perform the updates on the tuples for which local transactions have locks. Hence, for each such tuple X, T_i requests again a lock, and if the local transaction T_k still holds it (it might have aborted in between), T_i sends an abort request to T_k. T_k aborts and releases the lock which is directly granted to T_i (different to SI-P which wakes up all waiting transactions). Now T_i can perform the update.

Commit Phase: After all updates are performed, T_i updates $clog$ and releases all locks, waking up transactions.

Note that if there is no lock active on a tuple then an update is performed at the same time as the version check, avoiding to access a tuple twice whenever possible. Note also that there is no version check upon writeset delivery of a local transaction. If there is any conflict between a local transaction T_i and a remote transaction T_j whose writeset is delivered before T_i's writeset, T_i would have been aborted by T_j in T_j's late execution phase. Hence, upon delivering the writeset for T_i, if it is still alive, it has already passed the version check implicitly. Note also that a local transaction can commit when its writeset is locally delivered. The GCS guarantees that the writeset will be delivered, and hence, eventually executed at all replicas.

We will first give a simple example of execution as depicted in Figure 2. An initial transaction T_0 with GID=0 has updated X at replicas R1 and R2. Now assume T_1 is local on R1 updating X, and T_2 is local on R2 and also updates X. Both multicast their writesets which are delivered T_1 before T_2 at both replicas with GID=1 and GID=2 respectively. Both contain new versions of X together with GID=0. At R1, T_1 simply commits without further check. At R2, the valid version was created by T_0 with GID=0. Hence, the check succeeds. However, local transaction T_2 holds a lock and is aborted. When T_2 is delivered at $R1$, there is a valid version created by transaction T_1 with GID=1. T_2's writeset, however, piggybacks GID=0. Hence, the conflict check fails, and T_2 aborts. At R2, T_2's writeset is discarded because T_2 already aborted. If there were a third replica R3, T_1 would succeed the version check while T_2 would fail because T_1's version would be valid at the time of version check. That is, all replicas commit T_1 and abort T_2. Note that the GID counter is increased even when writesets of aborting/aborted transactions are delivered to guarantee that all replicas produce the same $GIDs$.

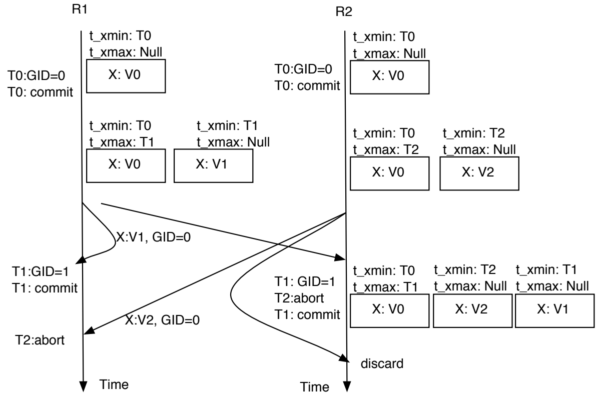

Figure 2. Example execution of SI-PR

We now want to give an example why a remote transaction should not immediately abort a local transaction whose writeset has been sent but not yet delivered. Assume three replicas and objects X and Y (with valid version referring to GID=0). Now assume R1 has transaction T_1 updating X, R2 has transaction T_2 updating first Y and then X, and R3 has transaction T_3 updating Y. All three send their writesets concurrently. Assume the delivery order is T_1 (GID=1) before T_2 (GID=2) before T_3 (GID=3). With this, T_1 should commit, T_2 should abort because it is concurrent to T_1 and also updates X, and T_3 should commit since it does not conflict with T_1. At R1, this will happen since at delivery of T_2, T_1 has committed and created a valid version of X labeled with GID=1. Hence T_2 will abort, and as a result T_3 can again commit. At R2, T_2 aborts upon delivery of T_1, and hence, T_1 and T_3 can commit. At R3, T_1 commits. When T_2 gets delivered, assume that it performs first the check on Y. It passes the check, but local T_3 holds the lock. T_2 should not abort T_3 because it will later fail the conflict test on X and will eventually abort. If it aborted T_3 prematurely, T_3 would commit at R1 and R2 but abort at its local replica. That is, a remote transaction T_i should only abort a local transaction T_j once T_i knows it will commit.

Correctness For space reasons, we do not provide a full proof of correctness. Instead, we will show informally that a transaction T_i either commits at all replicas or at none. Let T_i be local at R and remote at R'. (i) If T_i aborts at R before sending the writeset, then R' will not even receive a writeset. (ii) Assume T_i has already multicast its writeset and is later aborted at R. Since T_i has already completed execution before sending the writeset, this can only happen when the writeset of a remote transaction T_j is delivered before T_i's writeset, conflicts with T_i on a tuple X and commits. Without loss of generality, assume there is no other transaction T_k after that that also updates X. We show now that T_i will not pass the version check at R'. When T_i starts execution at R', R' has already committed T_j (serial commit/execution of writesets). The version of X created by T_j will be the valid version against which T_i will perform its version check. Of course, this is not the version that T_i accessed when it performed the update on local replica R (because T_i executed before T_j started at R). Hence, a mismatch between GIDs will be determined at R' and T_i will abort. (iii) Assume T_i commits at its local replica R. This means, no conflicting transaction's writeset is delivered between T_i's execution and its writeset delivery. Hence, when T_i is executed at R', the valid version for tuple X at R' was created by the same transaction that created the version T_i accessed during its local execution at R and the two GIDs are the same. As a result, the check succeeds.

Executing remote transactions concurrently A local transaction on replica R might be indirectly delayed by the serial execution of remote transactions executing locally on R. [22] proposes to only execute the version check for remote transactions serially and then execute non-conflicting transactions in parallel. This allows a local transaction to commit before previously received, non-conflicting remote transactions have finished execution. The problem is that the version check requires to retrieve all valid versions which is the most time consuming part of an update.

Instead, we suggest to perform what we call a *pre-lock phase* which requires a special lock table LT (independent from PostreSQL's locking). For space reasons, we only outline the approach. Upon delivery of a writeset WS_i, lock requests are included into LT for all tuples in WS_i in an atomic step (i.e., pre-lock phases of all transactions are serial). Once all locks are appended (which is fast), the next writeset can be processed. Once all locks for a local transaction T_i are granted, T_i can commit and release the locks. When all locks for a remote transaction T_i are granted, we continue with the version check and early execution phase. We release the locks in LT once T_i fails or completely commits. If T_i aborts a local transaction T_j who has already performed the pre-lock phase (but still waits for the locks), T_j's locks are removed from LT. If T_j had already sent the writeset but it was not yet delivered, we have to catch and discard the writeset.

For instance, if we receive the writeset for T_i accessing X, and then for T_j accessing Y, they do not conflict, and hence, acquire both the locks and can be executed and committed cocurrently. However, if both T_i and T_j access X, T_j must wait until T_i terminates and releases its locks so that operations are executed in the correct order. Note that even if a remote transaction has all locks in LT granted, it must perform the version check and hence, might fail.

Using this scheme the following can happen. If T_i and T_j do not conflict, one replica R might commit T_i, start a transaction T_k reading both X and Y, and then commit T_j, and another replica R' might commit T_j, start a transaction T_l reading both X and Y, and then commit T_i. Although SI is provided individually at each replica, a centralized system executing the same set of transactions could not have provided these different snapshots to T_k and T_l. Note that our previous solution does not have the problem since all replicas commit all transactions in the same order.

4.3. Recovery

When a crashed replica rejoins or a completely new replica joins an existing replica group, the GCS delivers a view change message to all replicas (including the joining replica). For any writeset message in the system, it is either delivered before the view change message to all old replicas or after the view change message and then also to the joining replica. Recovery now requires a peer replica to provide the joining replica with the database state including all writesets delivered before the view change message.

If a crashed replica rejoins, it looks into its internal table to determine the GID of the last transaction it committed. The peer has to send the writesets, assembled from its log, of all committed transactions with higher GIDs up to the last writeset before the view change. The joining replica applies them serially, creating a new transaction for each writeset, and maintaining the GIDs in its internal table accordingly. No version check is necessary because only writesets of committed transactions are sent. During this recovery procedure the other replicas can continue executing transactions. The new replica will receive their writesets and buffer them. Once recovery has finished, it will execute the buffered and newly arriving writesets using the SI-PR protocol. Since all GIDs are correctly maintained, it will make the correct commit/abort decisions for these writesets. It can also start local transactions.

For a completely new replica, the peer will first apply all writesets that were delivered before the view change message. Then, it starts a long read transaction that reads the entire database. For each tuple, the visible version is transferred to the new replica together with the GID of the transaction that created it. At the new replica, the tuple versions are installed with the necessary GID information. After that the new replica switches to normal processing as above.

At restart after a total failure, the replica with the largest GID in its internal table will start a new replica group. The others can join and perform recovery as described above.

5. Implementation

Postgres-R(SI) uses the same basic architecture as Postgres-R [21], and the writeset functionality has changed little. However, transaction execution and the interaction between components has changed considerably.

5.1. Writesets

Write operations are bundled into a single writeset message during normal processing. For each SQL DML statement (i.e., update, delete, insert), the writeset contains for each changed tuple (i) the GID of the transaction whose version V was copied, (ii) all modified attribute values and the corresponding attribute identifiers, and (iii) the primary key values of the tuple. For SQL DDL statements (e.g., create table, create function, etc.) the writeset simply contains the query text [3]. Remote replicas process the statements in the order they appear in the writeset. For DDL statements, the execution path is the same as it is for a local transaction (parser, planner, etc.). For DML statements, for each tuple to be changed, the remote backend retrieves directly the valid version of the tuple using the primary key index skipping most of the normal planning and execution steps.

5.2. Architecture

Figure 3 depicts the architecture of Postgres-R(SI). In PostgreSQL, the postmaster process listens for a connection request from a client, and then creates a dedicated backend process which will connect to the client and execute its transactions. Postgres-R(SI) extends PostgreSQL with three new processes: *remote backend*, *replication manager* and *communication manager*. The original backends are now called *local backends*, and execute local transactions. A remote backend processes the writesets of remote transactions. The communication manager's only purpose is to hide the details of the GCS (currently Spread [32]). We will not further mention it. The replication manager (RM) is the coordinator of the replica control algorithm.

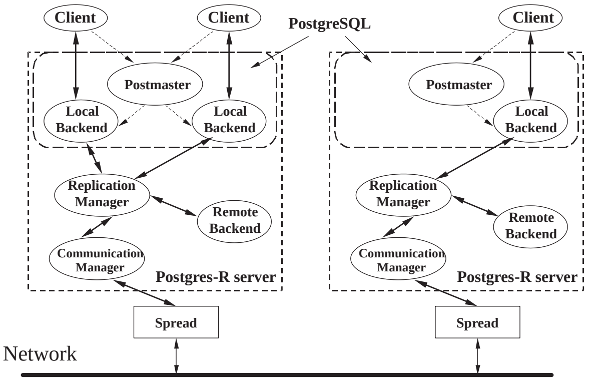

Figure 3. Architecture of Postgres-R

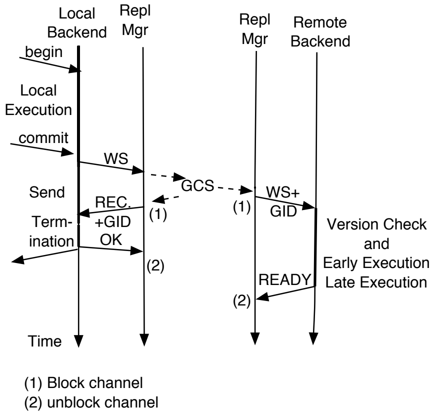

Figure 4. Transaction Execution

5.3. Transaction Execution

Figure 4 describes the execution of a successfully committing transaction T. T and the local execution phase starts at the local backend upon receiving a begin statement from the client (PostgreSQL also supports individual SQL statements to be transactions). During query execution, if there is any update performed, the changed tuples are added to the writeset. Upon receiving a commit request, the local backend checks the writeset. If it is empty, T commits. Otherwise, the writeset will be sent to the RM which multicasts it using the total order multicast of the GCS. Upon receiving this message from the GCS, each RM first generates the GID. If the writeset is for a local transaction, the GID is sent in a RECEIVED message to the local backend. If it is for a remote transaction, the GID is sent together with the writeset to the remote backend. To guarantee serial execution of writesets, the RM stops listening for any messages from the GCS. When the local backend gets the RECEIVED message, it commits T, and sends an OK to the

[3] Not all DDL statements will be replicated. [10] discusses which statements to replicate, which to only executed at the replica they are submitted to, or which to disallow in a replicated system.

RM. When the remote backend receives a writeset, it starts T's execution. Once it terminates it sends a READY message to the RM. Upon receiving the OK or READY message, the RM accepts the next writeset from the GCS.

Both local and remote transactions might abort. If a remote transaction fails the version check, its READY message to the RM contains the abort information. If a local transaction aborts before sending the writeset it does not need to notify the RM. If it aborts after sending the writeset, it must inform the RM with an ABORT message so that the RM can discard the writeset when it receives it from the GCS. Such an abort can only be triggered by a remote transaction since local execution has already finished. When a remote transaction aborts one or more local transactions, the RM might receive the READY message and corresponding ABORT messages in any order. It will wait until it has received all such messages before accepting the next writeset from the GCS. Furthermore, the RM might receive the ABORT message of a transaction T_i and then a new writeset for a new transaction T_j from the same local backend before receiving T_i's writeset from the GCS. Hence, the RM must remember aborted transactions for each backend.

The current system does not allow concurrent execution of non-conflicting remote transactions. The pre-lock phase could be either implemented in the RM or done by the backends using a lock table in shared memory. The latter approach could reuse lock functions provided by PostgreSQL.

5.4. Global Transaction Identifiers

The RM generates GIDs. It keeps a GID counter that is increased every time a writeset is received from the GCS, and assigns it to the corresponding transaction[4].

Both local and remote transactions need an efficient way to find the GID for a given TID. They have to find the GID corresponding to the t_xmin of the valid version V of a tuple X they want to update. A local transaction needs to include the GID in the writeset, a remote transaction must compare it with the GID included in the writeset.

There are several possibilties to store GID/TID pairs. The GID can be added to each tuple in addition to the TID. This would require to access tuples of local transactions a second time, since the GID is only generated after writeset delivery. Alternatively, we can add it to either PostgreSQL's standard REDO log or the $clog$. This would require to considerably change existing structures. Hence, we have chosen a third alternative, and created a new PostgreSQL system table `pg_gid` with GID and TID as attributes. PostgreSQL provides a cache for system tables with very fast access. Note that `pg_gid` is not replicated since it contains different data in the different replicas[5]. We provide abstract functions for accessing `pg_gid` which allows an easy migration to a different implementation.

5.5. Aborting local Transactions

A remote transaction must be able to force local transactions to abort. For that, we first have to understand how PostgreSQL handles aborts. In the simple cases, a transaction aborts due to failing a version check, a deadlock, or because the client submits an abort command. The backend simply executes an abort routine, informs the client about the abort, and awaits the next input. In a more difficult scenario, the backend receives a query–cancel signal (the client has sent a cancel connection request in the middle of execution) or shutdown signal from the postmaster. When the backend receives such signal, it can be in the middle of executing any function, e.g., a low-level subroutine manipulating main memory data structures. There is no guarantee that the database would remain consistent if the transaction would abort immediately. Therefore, PostgreSQL declares some variables indicating the state of the backend. Only if the backend is in a 'safe' state, the signal is treated as an exception leading to the immediate execution of the abort routine. If it is not safe, a flag is set. During query processing, the backend checks the flag at some safe spots and executes the abort routine if the flag is set.

If a remote transaction now wants to abort a local transaction it has to inform the local transaction. We decided to use signals for that. However, we must again make sure that the local transaction only aborts in a safe state. Additionally to the different states in the centralized system, we have several new situations. First, when the writeset is sent, the backend will wait for the writeset delivery confirmation. This is a safe state and we want the transaction to abort immediately. Second, when the local transaction sends the writeset to the RM, we do not want it to be interrupted in the middle of the transmission to avoid having partial messages left in the socket between backend and RM. Third, a transaction cannot be aborted when it is waiting for the input from a client or it is in the middle of input transmission. Otherwise, partial client requests might be left in the communication channel. Note that the original PostgreSQL allows to abort a transaction when the backend is waiting for client input only in case that there is a disconnection request from the client or the database is going to shutdown, in which case such partial results do not play a role.

In order to handle the last two cases, additional variables and flags have been added to the transaction context indicating whether and when a local transaction can abort. Note that it is possible that a remote transaction sends an abort

[4] One could have continuous GIDs by decreasing the counter when aborts occur. However, this is not needed for correctness.

[5] None of the system tables is replicated.

signal to an aborting transaction. In this case, PostgreSQL guarantees to abort the transaction only once.

5.6. Locking

Another problem to be solved is how a local transaction, aborted by a remote transaction, hands over a lock to the remote transaction. In theory, this should not be a problem. In PostgreSQL, when a process joins the waiting queue, it is normally appended to the end of the queue. We could simply adjust this procedure and put the lock request of the remote transaction at the head of the waiting queue.

However, in PostgreSQL, upon releasing a lock, all waiting processes are woken up. Although they are woken up in the order in which they are waiting, this does not guarantee that the lock is actually granted to the first one in the queue due to possible race conditions of UNIX process scheduling. Hence, we had to change the lock release procedure slightly. If a remote transaction T_i requests a lock held by local transaction T_j, we put the lock request of T_i at the head of the waiting queue, and send an abort signal to T_j. When T_j aborts and releases the lock, it determines that the first waiting transaction T_i is a remote transaction, and only wakes up T_i. The rest of the waiting queue is passed to T_i. When T_i receives the lock, it wakes up the rest of the processes in the waiting queue (in order to continue with the standard PostgreSQL procedure).

Note also that remote transactions can only abort when failing a version check. Hence, remote transactions should never invoke the deadlock detection routine. We achieve this by not setting the timer for the deadlock detection.

5.7. Implementation Overhead

Code was mainly added in form of new components (replication manager and remote backend). The remote backend has a different execution logic than the local backend but otherwise mainly calls PostgreSQL internal functions. Changes and adjustments to existing PostgreSQL code (e.g., locking, local commit, etc.), were not very large. We are currently adjusting a recovery component developed for a primary copy approach [10] to work with the new replica control algorithm.

6. Evaluation and Discussion

We evaluated the performance of Postgres-R(SI) using various tests. The first test uses the TPC-W benchmark to evaluate our system on a real application. The others all test special cases and use a database consisting of 10 tables with 10000 tuples each. All experiments are performed on a cluster of PCs (2.66 GHz Pentium 4 with 512 M RAM) running RedHat Linux. For each experiment, we run at least 20000 transactions to achieve stable results. Each test run has a fixed set of clients. A client submits a transaction and then sleeps (thinks) for a certain time in order to achieve the desired system wide throughput measured in transactions per second (tps).

6.1. TPC-W Benchmark

In order to evaluate our system against a real-world application, we performed our first exerpiment using the OSDL-DBT-1 benchmark [24]. It is a simplified version of the TPC-W benchmark [13] simulating an online bookstore. It has three different workload types by varying the ratio of browsing to buying transactions. We have chosen the browsing workload, which contains 80% browsing and 20% ordering transactions. We have set up a two-tier testbed where the OSDL-DBT-1 driver is the front-tier which directly connects to the database. There are 8 tables in the schema. The database size is determined by the items and clients in the system. We use a very small configuration with only 1000 items and 40 clients. Larger sizes will only decrease conflict rates and increase disk I/O which will favor the replicated approach. We performed the experiment with a fixed number of 40 client connections. The number of clients on each server and the load on each client is evenly distributed.

We run the experiment with a centralized, non-replicated server, and then with 5 and 10 replicas. Figure 5 shows the client response time for browsing transactions, and Figure 6 shows the response time for ordering transactions when we increase the overall load to the system. For all graphs, the response time increases with increasing load since more transactions concurrently compete for resources. The response time of the centralized system is much worse than our replicated configuration, and can achieve a much lower maximum throughput. The reason is that the server is overloaded very fast while in the replicated systems read-only transactions are distributed among the replicas. Additionally, the centralized server has problems handling many clients. The 10-replica system has smaller response times than the 5-replica system for a given throughput because read-only transactions are distributed over even more replicas. The only exception are update transactions at 20 tps where the 5-replica system is better than 10 replicas. The reason might be that with 10 replicas, more update transactions are remote, and hence, it is more likely that a local update transaction has to wait for a remote transaction whose writeset is received earlier. At higher throughputs this disadvantage does not show because the 10-replica system is much less loaded. In these experiments, abort rates were always well below 1%, which shows that SI can handle real world conflict rates even for very small database sizes.

However, scalability is not unlimited. Updates have to be performed at all replicas. If the update load increases, each

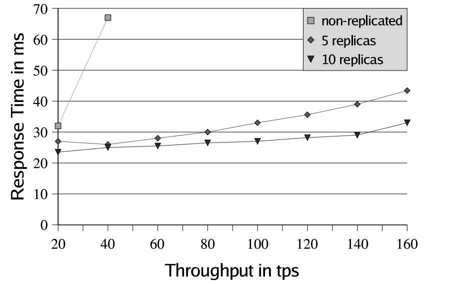

Figure 5. TPC-W: Browsing (read-only)

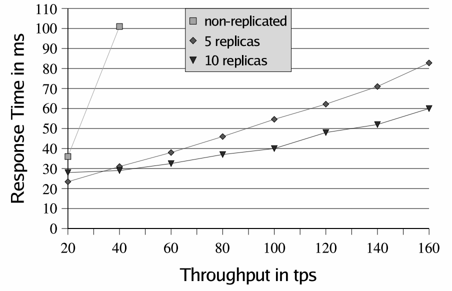

Figure 6. TPC-W: Ordering (update)

replica has less resources to execute queries. Hence, the performance gain from 5 to 10 replicas is not as big as from the non-replicated system to 5 replicas. More about this phenomena can be found in [20].

In summary, this experiment proves that the performance of our system is excellent for a real world situation where most of the transactions are read-only. Our replication solution performs better than a centralized approach by distributing the load and clients throughout the replicas in the system. Hence, eager update everywhere replication based on SI is feasible for real-world applications.

6.2. Replication Overhead

In order to evaluate the delay incurred by replication, we tested a system with a single client on our 10 table database submitting transactions that update 10 random tuples. A replicated system of 5 replicas needs 5.5 ms more to finish the transaction than a non-replicated PostgreSQL server. We consider this quite acceptable considering that it includes the overhead of generating the writeset and forwarding it through the replication manager and Spread.

6.3. Update Intensive Workloads

The third experiment has a closer look at the behavior for update transactions at higher loads. The workload consists of 100% update transactions. A transaction performs 10 updates, each on a random tuple. This is a worst case scenario where we can expect the replicated system to always perform worse than a central system. The setup includes a non-replicated PostgreSQL and a replicated system with 5 nodes, and either 5 or 20 clients in the system.

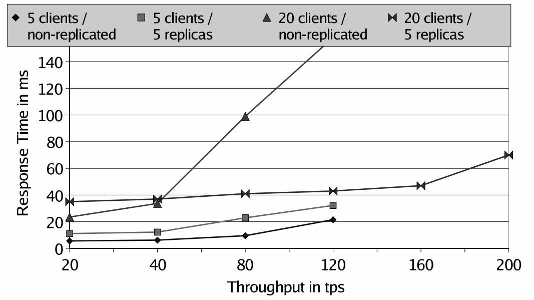

Figure 7. Update Workload: Response Time

Figure 7 shows the response time with increasing load. For 5 clients, the central system has always considerably faster response times than the replicated system until the maximum throughput for 5 clients is achieved (no sleep time between transaction submission). This shows the replication overhead for update transactions similar to the previous experiment. For 20 clients, however, the results are very different. At low throughputs, the central system has faster response time. However, once the throughput passes 40 tps, the central system starts to be overloaded and experiences increasing response times while the response time in the replicated system remains low. Not shown in the figure, abort rates are between 1% and 1.5% for the replicated system, for the central system they start at 0.2% at 20 tps and increase to nearly 7% at 120 tps due to the increase in response time. The main reason for the sharply increasing response times and abort rates is that the central system has difficulties to manage 20 clients. Although the clients are often idle (waiting between two submissions), they put a considerable administrative burden on the system. This shows that replication might even pay off in update intensive workloads in cases it enables to distribute other kinds of work, for instance, client management.

6.4. Comparison with other Approaches

We cannot provide direct comparison with the original Postgres-R based on locking because the underlying systems, version 6 vs. version 7, differ extremely in their buffer management and other modules. In general, however, the relative performance of both approaches is similar. This proves that the general approach of executing transactions locally, multicasting writesets at the end of the transaction and applying them efficiently, works well in a LAN setting.

We think that a direct comparison with middleware based solutions, even if they have been evaluated using the TPC-W benchmark [1, 29, 2], is unfair since the setups are always quite different.

7. Conclusion

This paper presents the design, implementation and integration of an eager, update everywhere database replication approach based on snapshot isolation within the open-source database system PostgreSQL. Replication is transparent to clients which can submit transactions to any replica, and perceive the same level of isolation as a centralized system. In fact, existing application code can remain completely unchanged (except of, maybe, a load-balancing routine, or a routine that will switch to another replica in case of a failure). Our experiments in a cluster of workstations show that the replicated system provides scalability and keeps response times small. Its integration as an additional module into a database system makes its installation and use simple. Hence this approach is well suited to migrate databases from overloaded centralized servers to a cluster architecture. We are currently making our solution open-source under the pgreplication project of PostgreSQL at http://gborg.postgresql.org/project/pgreplication/.

References

[1] C. Amza, A. L. Cox, and W. Zwaenepoel. Distributed versioning: Consistent replication for scaling back-end databases of dynamic content web sites. In *Middleware'03*.

[2] C. Amza, A. L. Cox, and W. Zwaenepoel. A comparative evaluation of transparent scaling techniques for dynamic content servers. In *Int. Conf. on Data Engineering*, 2005.

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

[4] H. Berenson, P. A. Bernstein, J. Gray, J. Melton, E. J. O'Neil, and P. E. O'Neil. A critique of ANSI SQL isolation levels. In *ACM SIGMOD Conf.*, 1995.

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

[6] K. Böhm, T. Grabs, U. Röhm, and H.-J. Schek. Evaluating the coordination overhead of synchronous replica maintenance in a cluster of databases. In *Euro-Par*, 2000.

[7] Y. Breitbart, R. Komondoor, R. Rastogi, S. Seshadri, and A. Silberschatz. Update propagation protocols for replicated databases. In *ACM SIGMOD Conf.*, 1999.

[8] E. Cecchet, J. Marguerite, and W. Zwaenepoel. C-JDBC: Flexible database clustering middleware. In *USENIX Annual Technical Conference*, 2004.

[9] G. V. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: A comprehensive study. *ACM Computer Surveys*, 33(4), 2001.

[10] M. Chouk. Master–slave replication, failover and distributed recovery in PostgreSQL database. Master's thesis, McGill University, June 2003.

[11] B. S. Corporation. Interbase Documentation, 2004.

[12] O. Corporation. Oracle 9i Replication, June 2001.

[13] T. P. P. Council. TPC Benchmark W, 2000.

[14] K. Daudjee and K. Salem. Lazy database replication with ordering guarantees. In *ICDE*, 2004.

[15] R. Goldring. A discussion of relational database replication technology. *InfoDB*, 8(1), 1994.

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

[17] T. P. G. D. Group. PostgreSQL 7.2 Documentation, 2001.

[18] J. Holliday, D. Agrawal, and A. E. Abbadi. The performance of database replication with group communication. In *Int. Symp. on Fault-tolerant Computing*, 1999.

[19] J. Holliday, R. Steinke, D. Agrawal, and A. Abbadi. Epidemic algorithms for replicated databases. *TKDE*, 15(5), 2003.

[20] R. Jiménez-Peris, M. Patiño-Martínez, G. Alonso, and B. Kemme. Are quorums an alternative for data replication. *ACM Transactions on Database Systems*, 28(3), 2003.

[21] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-R, a new way to implement database replication. In *Int. Conf. on Very Large Databases*, 2000.

[22] B. Kemme and G. Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Transactions on Database Systems*, 25(3), 2000.

[23] J. M. Milan-Franco, R. Jiménez-Peris, M. Patiño-Martínez, and B. Kemme. Adaptive middleware for data replication. In *Middleware*, 2004.

[24] Open Source Development Lab. Descriptions and Documentation of OSDL-DBT-1, 2002.

[25] E. Pacitti, P. Minet, and E. Simon. Replica consistency in lazy master replicated databases. *Distributed and Parallel Databases*, 9(3), 2001.

[26] E. Pacitti and E. Simon. Update propagation strategies to improve freshness in lazy master replicated databases. *VLDB Journal*, 8(3), 2000.

[27] F. Pedone, R. Guerraoui, and A. Schiper. Exploiting atomic broadcast in replicated databases. In D. J. Pritchard and J. Reeve, editors, *Euro-Par*, 1998.

[28] PeerDirect. Overview and comparison of data replication architectures. White Paper, Nov. 2002.

[29] C. Plattner and G. Alonso. Ganymed: Scalable replication for transactional web applications. In *Middleware*, 2004.

[30] U. Röhm, K. Böhm, H.-J. Schek, and H. Schuldt. FAS - a freshness-sensitive coordination middleware for a cluster of olap components. In *VLDB*, 2002.

[31] R. Schenkel, G. Weikum, N. Weienberg, and X. Wu. Federated transaction management with snapshot isolation. In *Transactions and Database Dynamics, Int. Worksh, on Found. of Models and Lang. for Data and Objects*, 1999.

[32] Spread. homepage: http://www.spread.org/.