Integrating Synchronous Update Everywhere Replication into the PostgreSQL Database System based on Snapshot Isolation

by:
Shuqing Wu

A Thesis
Submitted in Partial Fulfillment of the Requirements
for the Degree of Master of Science
at
McGill University
Montréal, Canada

October 2004

Your file Votre référence
ISBN: 0-494-06473-0
Our file Notre référence
ISBN: 0-494-06473-0

Abstract

This thesis presents the integration of a synchronous, update everywhere replication protocol into the relational database system PostgreSQL. This work is based on previous work which integrated replication into PostgreSQL 6.4 using strict 2–phase–locking for concurrency control. In this thesis, we migrated the approach to PostgreSQL 7.2 which uses a multi–version concurrency control mechanism providing the isolation level Snapshot Isolation. This required a complete redesign of the replica control component. With our approach, transactions can be submitted to any replica. This replica executes the transaction locally and multicasts the updates to the other replicas. The combined concurrency and replica control components guarantee that concurrent updates are serialized in the same order at all replicas, providing the same isolation level as a non–replicated system. The thesis also presents a performance evaluation showing that our approach has little overhead and provides scalability up to 20 replicas.

Résumé

Nous présentons à l'intérieur de cette thèse l'intégration d'un protocole de réplication synchrone avec mise à jour globale (update-everywhere) au système de gestion de base de données relationnelle PostgreSQL. Les travaux précédents sur le sujet, sur lesquels nous nous sommes basés, intégraient un protocole de réplication à PostgreSQL 6.4 en utilisant le protocole de verrouillage à deux phases strict (strict 2PL) comme contrôle d'accès simultané. Dans cette thèse, nous avons plutôt concentré notre travail sur PostgreSQL 7.2, qui utilise un mécanisme de contrôle d'accès à versions multiples fournissant une isolation de niveau Snapshot. Ceci a donc requis une restructuration complète de la composante de contrôle de réplication. Avec notre approche, chaque transaction peut être soumise à n'importe quelle réplique. Cette dernière exécute la transaction et multidiffuse les mises à jour aux autres répliques. La combinaison des composantes de contrôle d'accès simultané et de réplication guarantissent que les mises à jour simultanées seront sérialisées dans le même ordre à chacune des répliques, fournissant ainsi le même niveau d'isolation que le système non répliqué. Nous évaluons aussi dans cette thèse la performance de notre approche et nous discutons des coûts additionnels qu'elle implique et des limites de son extensibilité.

Acknowledgements

I would like to gratefully acknowledge the enthusiastic supervision of Prof. Bettina Kemme during this work. Throughout my graduate study, she provided great teaching, sound advice and encouragement. Her patience is greatly appreciated.

I would like to thank several of my friends for helping with the preparation of this thesis. Thomas Feng, Zhan Yu, Lili Chen, Huaigu Wu, Yi Lin, Chenliang Sun and Yishen Liu have provided me a great environment to live and study.

I wish to thank my parents, Qiying Tong and Fukang Wu for their endless love and support throughout my live. I am also grateful to Qiongping Dong and Mianmo Li for their support.

I cannot end without thanking Fang Li, on whose constant encouragement and love I have relied in the last four years.

Contents

Chapter 1

Introduction

Storing, managing, and retrieving information are very crucial tasks for businesses and in our daily life. Databases allow us to achieve this easily and efficiently. In the last four decades, the use of databases has grown tremendously. Databases have become an important component in the development of a variety of applications. Fault tolerance and performance are two important features that must be provided by database systems. Using replication is one of the solutions. Replication is the technique that creates and maintains several copies of a database and stores them on different servers (nodes). Fault tolerance is achieved by providing access to different copies. When one of the nodes experiences a failure, the user application can keep functioning by connecting to other accessible servers. Performance, in terms of high throughput, low response time and good scalability, is gained by distributing the load submitted to the system over all copies. In a WAN environment, typically the nearest copy is accessed. However, achieving reasonable performance without compromising data consistency is a big challenge in a replicated database. The different copies of the database have to be kept consistent despite updates. This task is called replica control and requires coordination and communication among nodes. Some replication algorithms proposed in the research community are too complicated to be used in practice. Some others are trading correctness for performance.

Replica control solutions can be categorized to be either *synchronous* or *asynchronous* [17]. Using a synchronous approach, updates are propagated within the

transaction boundaries, i.e. before the transaction commits and the user is informed about the outcome. Using this approach, data consistency among replicas can be guaranteed. In contrast, updates are propagated after the transaction has committed in the asynchronous replication approach. This approach does not provide full fault tolerance, since transactions that committed locally but whose updates were not propagated before a crash are lost. Although only synchronous solutions can guarantee full data consistency, most of the commercial databases are using the asynchronous approach due to its better performance.

Replica control solutions can also be categorized to be either *master/slave* or *update everywhere*. Using the master/slave approach, also called *primary copy* approach, update operations can only be submitted to the master node (which then propagates them to the other nodes). In contrast, update everywhere allows any copy in the system to be updated. To preserve data consistency, efforts have to be made to coordinate operations submitted to different copies.

Replica control techniques often depend on the underlying concurrency control mechanism. The most popular concurrency control approach is *2–phase–locking*. Using 2–phase–locking, a transaction must acquire a lock on a data object before accessing the object and if one holds a lock allowing it to update the data object, no other lock is granted. Instead, a transaction requesting a lock has to wait until the granted lock is released. Using 2–phase–locking, a transaction is not allowed to acquire any further lock once it has released a lock. One problem of this approach is deadlock. Other concurrency control mechanisms keep several versions of a data object, allowing read operations to access older versions while writes create a new version. Depending on the particular concurrency control mechanism, this allows read and write operations on the same data object to run concurrently. PostgreSQL 7.2 and Oracle use such a multi–version concurrency control mechanism. However, they do not provide the standard isolation level *serializable* [7] as guaranteed by 2–phase–locking protocols, but a lower isolation level called snapshot isolation [7]. Therefore, we refer to them here as "snapshot isolation" concurrency control mechanisms.

In [22], the author presents a suite of synchronous and update everywhere database

replication protocols. These protocols use powerful multicast primitives to send updates to all replicas and to help determining the execution order. Furthermore, they attempt to keep message overhead small by sending all updates of a transaction in a single message. The approach is designed to work for LAN environments with a cluster of workstations holding the database replicas. One of the proposed algorithms is based on a multi–version concurrency control mechanism.

The object of this thesis is to integrate this algorithm into PostgreSQL 7.2. Our main challenge was that the rather abstract algorithm in [22] had to be adjusted considerably to fit the concrete implementation of snapshot isolation in PostgreSQL. Our implementation takes advantage of an existing replication architecture for PostgreSQL, called Postgres–R. Several versions of Postgres–R have been developed, one for locking based concurrency control (based on PostgreSQL 6.4), and one using a master/slave approach based on PostgreSQL 7.2. Our solution is a further step to provide a general update everywhere replication solution for PostgreSQL. We thoroughly evaluated our implementation showing that it is efficient and provides high throughput and good scalability.

The rest of the thesis is structured as follows: Chapter 2 provides necessary background information in regard to concurrency control, replica control and communication primitives. Chapter 3 presents the synchronous, update everywhere replication algorithm providing snapshot isolation proposed in [22] and redesigns it to be used in PostgreSQL. Chapter 4 shows the architecture of PostgreSQL and Postgres–R. Chapter 5 presents the implementation details. Chapter 6 shows the experiments and analyzes the results. Chapter 7 discusses possible optimizations and concludes the work.

Chapter 2

Background

In this chapter, we first introduce the main concepts behind transactions and provide an overview of several concurrency control mechanisms, in particular those providing the snapshot isolation level. Then, we provide an overview of replica control strategies. Finally, we introduce group communication systems, and outline how they can be used to support the tasks of replica control.

2.1 Transactions

A *transaction*T_i is a sequence of read operations $r_i(X)$ and write operations $w_i(X)$ on data objects X. Operations of a transaction build a logical unit of work from an application point of view. Several transactions can run concurrently in a database. Transactions must satisfy four primary properties, the so called $ACID$ properties. In our context, atomicity, isolation and durability are the most important. Atomicity means that either all or none of the operations of a transaction have effect on the database. If all operations succeed, we say the transaction commits (c_i), otherwise it aborts (a_i). In regard to isolation, several isolation levels are known. The most common isolation level is serializable: although transactions run concurrently, their execution has the same effect as a serial execution of these transactions. Durability finally guarantees that once a transaction has committed, its changes to the database are persistent despite system failures.

2.2 Concurrency Control

The *concurrency control* component provides isolation of concurrent transactions. It ensures that transactions do not interfere with one another and guides against incorrect results. The concurrency control component orders *conflicting* operations of different transactions. Two operations conflict if they access the same data object and at least one of them is a write operation. To guarantee serializable execution, the combined effect of interleaving operations of a set of transactions must be the same as if the transactions had executed one at a time in some serial execution.

There are three common concurrency control techniques presented in the literature: *locking*, *optimistic concurrency control* and *timestamp ordering* [13]. Using locking, a transaction that wants to access a data object, has to first acquire a lock for this object. To read an object, a transaction has to acquire a shared lock. When it wants to write an object, it has to acquire an exclusive lock. Several shared locks can be granted at the same time on an object. But if an exclusive lock on an object has been granted, no further lock (shared or exclusive lock) will be granted on the same object. All transactions requesting locks on that object must wait until the exclusive lock is released. In order to guarantee serializability, *2–phase–locking* ($2PL$) is used. In 2PL, the locking period for a transaction can be divided into a *growing* phase and a *shrinking* phase. During the growing phase, the transaction can acquire locks. Once the transaction releases the first lock, it enters the shrinking phase, and may not obtain any new locks. If all transactions follow the 2PL principle, their interleaved execution is guaranteed to be equivalent to a serial execution. The problem of this mechanism is that it is subject to deadlocks. Another potential problem is that *dirty read* is possible. When a transaction is in the shrinking phase and has released an exclusive lock, the changes made by this transaction are visible to other transactions. If the transaction aborts, these changes will be undone. However, other transactions might have seen the changes. This is called a dirty read. The consequence is that the transactions which see the changes also have to abort. This causes *cascading–abort*. To prevent this problem, we can use *strict* $2PL$. Using strict 2PL, exclusive locks are held until the transaction commits or aborts. The drawback of this approach is less

concurrency.

Locking oriented mechanisms, likes 2PL, are *pessimistic* concurrency control algorithms. Conflicts are detected as soon as they occur and resolved by blocking the execution of some of the transactions. In contrast, optimistic concurrency control algorithms detect conflicts at transaction commit time. This approach is based on the assumption that conflicts are rare. Transactions in an optimistic concurrency control algorithm have three phases: *read phase*, *validation phase* and *write phase* (only for transactions with write operation). In the read phase, whenever an operation accesses a data object X, the transaction creates its own copy (shadow copy) and works on the shadow copy from then on. Upon committing, the database performs a check (validation) to discover if there is any conflict with other concurrent transactions. The conflicts are resolved by aborting one transaction. The detail of the checks and the decision which transaction to abort in case of conflict depends on the particular algorithm. Many alternatives are possible such as forward validation or backward validation [19]. In forward validation, if the transaction conflicts with any other currently active transaction, it is aborted. In backward validation, it is aborted if it conflicts with any transaction that committed after the transaction to be validated started. If the transaction passes the validation phase, the changes will be written to the database and the transaction commits. There are several circumstances where optimistic concurrency control has better performance than 2PL: the large majority of the operations are read operations, the conflict rate is low, and transactions are short. Optimistic algorithms avoid deadlock and allow more concurrency. However, this approach has high abort rates.

Another concurrency control mechanism presented in the literature is timestamp ordering. Using timestamp ordering, all conflicting operations are executed in timestamp order. Each transaction is assigned a unique *timestamp* value (TS) when it starts. Every operation in a transaction is using this value. And each data object is assigned a read timestamp and a write timestamp which is the timestamp of the transaction that was the last to read or write the object. Timestamp ordering concurrency control obeys the following rules:

- A read operation of a transaction with timestamp TS can read an object whose

write timestamp is smaller than TS or equal to TS.

- A write operation of a transaction with timestamp TS can write an object whose both timestamps are smaller or equal to TS.

If the timestamp(s) of the object are already too large, the transaction must abort. With this, serializable execution is guaranteed. Since transactions never wait, the timestamp ordering mechanism is deadlock free. The disadvantage of this approach is a possibly high abort rate, since transactions are not blocked but immediately aborted. It can also cause cascading aborts. There is an improved version of the timestamp ordering approach, called *multi–version timestamp ordering*, which allows each transaction to create a new version of the object. This increases the concurrency level considerably by redirecting read operations to older versions instead of aborting them. In general, there exist many variations of concurrency control that use the concepts of locking, optimism, timestamps and multi–version databases.

So far, we have only looked at concurrency control mechanisms that provide serializability, that is, they produce executions that are equivalent to a serial execution. However, in practice, lower correctness criteria are often used that are defined in terms of isolation levels.

2.3 Snapshot Isolation (SI)

2.3.1 Isolation Levels

An isolation level describes the degree to which the data being updated by one transaction is visible to other transactions, thus allows application designers to trade off concurrency for correctness. ANSI SQL–92 specifies four isolation levels: *read uncommitted*, *read committed*, *repeatable read*, *serializable*. These isolation levels have been defined assuming lock–based concurrency control. They are defined in terms of three phenomena: *dirty read*, *non–repeatable read* and *phantom*. Each of these phenomena violates serializability to a certain degree. Each isolation level is characterized by ruling out some of the phenomena. [6] pointed out that the three ANSI phenomena are incomplete and ambiguous. To fix the weakness of the ANSI

Isolation Level	Dirty Read	Cursor Lost Update	Fuzzy Read	Phantom	Read Skew	Write Skew
Read Uncommitted	P	P	P	P	P	P
Read Committed	N	P	P	P	P	P
Cursor Stability	N	N	N	P	P	N
Repeatable Read	N	N	N	P	N	N
Snapshot	N	N	N	N	N	P
Serializable	N	N	N	N	N	N

P - Possible N - Not Possible

Table 2.1: Isolation Level Specification of [6]

SQL–92 specification, [6] introduced five new phenomena: *cursor lost update*, *lost update*, *fuzzy read*, *read skew* and *write skew*. Furthermore *snapshotisolation* (SI) and *cursorstability* are defined as two new isolation levels. Earlier discussions on variations of snapshot isolation can be found in [1]. Table 2.1 provides an overview of which isolation level avoids which phenomena.

2.3.2 Snapshot Isolation

In this thesis, we focus on snapshot isolation. SI can be implemented via a multi–version database system [1]. In a multi–version database system, each update creates a new version of a data object. Assume object versions are labelled with the transaction that created them and each transaction T_i is tagged with $TS_i(BOT)$ (begin of transaction) and $TS_i(EOT)$ (end of transaction) timestamps, e.g. physical time or an increasing counter. We say that two transactions T_i and T_j are concurrent if neither $TS_i(EOT) < TS_j(BOT)$ nor $TS_j(EOT) < TS_i(BOT)$. With this, a concurrency control protocol provides SI if:

1. Read operation r_i(X) of transaction T_i reads from the most recent committed version of X as of the time of the begin of transaction T_i. That is, T_i reads from a snapshot of the database.
2. Write action w_j(X) of a transaction T_j is invisible to a concurrent transaction T_i.
3. Write action w_i(X) will be reflected in the snapshot for T_i.
4. If two concurrent transactions T_i and T_j write a common data object X, at least

one of them must abort. For instance, if T_i commits, T_j must abort.

This approach effectively separates read and write operations. Read operations, never blocking or being blocked, do not conflict with write operations. Therefore read-only transactions execute concurrently with read–write transactions without any interference. Contention is significantly reduced. Obviously, if long-running transactions are read–only and update transactions are short, SI should give high performance. As a result, SI is a popular isolation level offered in centralized database management systems. For example, Borlands InterBase [11], PostgreSQL [18] and Oracle [12] offer SI.

SI avoids almost all of the phenomena. However, SI does not ensure that all executions are serializable. In particular, SI allows the write skew phenomenon. Suppose we have data objects X=50, and Y=50 with a constraint that $X + Y > 0$. Assume we have two transactions T_1 and T_2. T_1 subtracts 60 from Y, and T_2 subtracts 60 from X. Before performing the subtraction, both transactions need to read X and Y in order to check whether such a subtraction can be performed considering the constraint. We can have the following scenario: r_1[X=50] r_1[Y=50] r_2[Y=50] r_2[Y=50] w_1[Y=-10] w_2[X=-10] (c_1 c_2) This execution is possible under SI. However it is not serializable. The reason is that T_1 and T_2 read the same state of the database fulfilling the constraint but change different data objects. Since conflicts are only detected when transactions change the same object, each transaction is not aware of the state change made by the other transaction. [1] Note that, because of the potential problem of write skew, application designers must be aware of the nature of the SI and guard against the constraint violation.

2.4 Replica Control

Replication, as one of the key techniques to improve performance and fault–tolerance in database systems, has been a hot topic in the distributed computing and database

[1] Interestingly, since SI avoids all three phenomena described by ANSI SQL–92 standard, Oracle and PostgreSQL actually claim that their concurrency control mechanism using SI guarantees serializable executions.

management research field in the last two decades. We can find the fundamental theories related to database replication in [7]. Today, almost every major database vendor has a replication solution of one kind or another. However, only few of those solutions meet the correctness criteria described in [7]. A replicated database system should behave like a one–copy database (non–replicated), i.e. replication should be transparent to the user. That is, the execution over several copies should be conceptually equivalent to a serializable execution over a one–copy database. This is referred to as *one–copy–serializability.*

2.4.1 Categorizing Replication Strategies

As described in [17], replica control techniques can be categorized by **when** updates are propagated. A *synchronous* replication model, also called *eager* replication, propagates the changes before the transaction commits. In contrast, in the *asynchronous* replication model, also called *lazy* replication, changes made by a transaction are propagated to other nodes only after the transaction has committed. The potential problem of this model is that when a node crashes after it commits a transaction, but before it propagates the changes, the transaction is lost. From a conceptual point of view, synchronous replication is preferable because it provides data consistency and system correctness. However, it has high communication overhead, which can increase response time. In contrast, the communication overhead in lazy replication can be hidden by implementing propagation as a concurrent background process. Hence the lazy replication has better response time and scalability. As a result, most of the commercial database replication solutions apply the lazy model [16]. However, there are cases in which synchronous database replication is still feasible. Since message delays are low in LAN's, synchronous replication protocols are practical in cluster computing when databases are replicated on cluster of workstations.

Replication techniques can also be divided by **where** updates can be submitted. In a *master/slave* approach updates can only be submitted on the master copy, in an *update everywhere* approach updates can be submitted to any replica. The master/slave approach, also called *primary copy* approach, can be implemented without

too many adjustments to the concurrency control algorithms. The master can execute as a centralized database system. When slaves receive the updates propagated by the master, they have to apply conflicting updates in the same order they were executed at the master. The master/slave approach introduces a potential bottleneck and single point of failure. The update everywhere approach, in contrast, is more flexible. However, update everywhere is much more complicated in the design and implementation.

If update everywhere is combined with asynchronous replication, it can happen that two transactions T_1 and T_2 concurrently update data object X on two different replicas and both commit locally not being aware of the conflict situation. This requires reconciliation to bring the data copies back to a consistent state which is far of being trivial. Combining update everywhere with synchronous replication requires a careful integration of replica control and concurrency control and might lead to a significant communication overhead among the replicas. The next section describes some traditional synchronous update everywhere solutions.

2.4.2 Traditional Solutions

Textbook replication solutions have been traditionally synchronous update everywhere [7]. In a basic *ROWA* (read–one/write–all) approach based on 2PL, a read operation is executed locally, a write is multicast to all replicas. All replicas acquire locks and execute the operations. A transaction can only commit when all write operations succeed on all replicas. When one replica fails, the system is blocked. *ROWAA* (read-one/write-all available) extends this approach and only requires to write all available copies.

One problem of this approach is that each transaction has to check its view of available replicas constantly to avoid one replica updating a copy which is not accessible to another transaction. This problem is caused by network partitions. To deal with network partitions, *quorum* protocols have been proposed. They require both read and write operations to access a quorum of copies. This approach does not scale for read–intensive applications. The poor response times (too much communication

and coordination) and the lack of scalability of these solutions made synchronous, update everywhere replication unattractive. Only few commercial databases are using this model. Instead, these mainly use master/slave replication to easily guarantee serializability, and asynchronous replication to avoid communication within transaction response time, hence sacrificing flexibility, consistency, and durability of transactions [16].

[17, 22] attempt to address the limitations of synchronous update everywhere textbook solutions and propose to take advantage of group communication systems (GCS) to help with replica control. GCS provide powerful multicast primitives that guarantee message delivery and order messages.

The next section describes group communication systems and their functionality in more detail. Only then will we be able to discuss this kind of replication approach in more detail.

2.5 Group communication systems (GCS)

Group communication systems (GCS) play a very important role in the development of distributed systems. Beginning with ISIS [8], and followed by many others, such as Horus [27], Transis [15] and Spread [2], GCS have a well–developed history, especially for LAN environments. A GCS is a framework that facilitates the task of constructing reliable and complex distributed applications. It gives the system designer a powerful set of abstractions upon which many different distributed applications can be built. GCS typically provide *multicast* and *membership* services. A set of processes or nodes form a group, and each group member can multicast messages to the group. The messages are delivered to each available group member (including the sender). A node can join or leave the group. If a node crashes, the GCS automatically removes the node from the group. All nodes of the new group configuration are informed about changes in the group configuration.

Typically, the semantics of the multicast primitives can be categorized by two parameters, the ordering semantics and reliability semantics. There are four message ordering semantics:

- *unordered*: No ordering guarantee. Each node can receive any message in any order.
- *FIFO*: Messages of each sender are delivered at each node in the order they were sent.
- *causal*: Messages sent by all senders are delivered in an order which preserves Lamport's happen-before relation [13].
- *total*: Messages sent by all senders are delivered in the same total order at all nodes.

There are three message reliability semantics:

- *unreliable*: Messages may be dropped or lost and will not be recovered.
- *reliable*: Whenever a node delivers a message, and this node does not fail for sufficiently long time, then all other group members will deliver the message unless they fail.
- *uniform–reliable*: Whenever a node delivers a message, all other nodes will deliver the message unless they fail (no exception).

Our implementation uses *Spread* [2], a well–known open–source GCS for both local and wide area networks. It outstands by its stability and rich group communication semantics. Spread offers powerful multicast primitives and a membership service with strong semantics.

Applications can multicast messages by selecting different ordering and reliability semantics. However, in Spread, not all reliability semantics can be combined with all ordering semantics. Spread supports five different types of service: unreliable/unordered, reliable/unordered, reliable/FIFO, reliable/causal, reliable/total (called *agreed* in Spread) and uniform-reliable/total (called *safe* in Spread).

In addition to the regular message service, Spread also maintains membership information about who is alive and reachable in the group with the same strong ordering and reliability semantics. When the GCS detects a crash, all surviving nodes will deliver a view change message informing about the new membership configuration. Spread supports the *extended virtual synchrony* model [24] which is the extension of the *virtual synchrony* model [9] to handle network partitions. Since our

work focuses on cluster databases in a LAN environment, the virtual synchrony model is sufficient. Virtual synchrony, which was introduced in ISIS, guarantees that membership changes within a group are observed in the same order by all the members that remain connected. Moreover, membership change messages are totally ordered with respect to all regular messages in the system. This means that every two processes that observe the same two consecutive membership changes, receive the same set of regular multicast messages between the two changes. This property ensures delivery atomicity with respect to views of membership in a group.

2.6 Replica Control Based on GCS

There have been many proposals suggesting to use GCS to support replica control [3, 20, 22]. In this thesis, we follow the approach of [22]. The basic idea is as follows. A transaction T_i can be submitted to any replica. This replica is T_is 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 (called 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 will 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 receive or has already received it. Hence, it is safe to commit/abort a transaction locally because the other replicas will do the same. Only one message is sent within the transaction boundaries. 2–phase–commit is avoided, and a transaction can commit locally without waiting that other replicas have executed the writeset. When replicas fail, the GCS informs the remaining replicas. They simply can continue as a smaller group. This approach avoids many of the limiations pointed out by [17], at least for local area networks.

Chapter 3

Replica and Concurrency Control providing SI

In this chapter, we present the replica control algorithm that we have implemented in PostgreSQL. Our description follows an incremental approach. In Section 3.1, we present an abstract centralized concurrency control algorithm providing SI. We call this algorithm SI–C (***c***entralized concurrency control providing SI). SI–C was the baseline for the replica control algorithm proposed in [22] which we present in Section 3.2. We refer to this algorithm as SI–R (concurrency and ***r***eplica control providing SI). However, we were not able to directly implement SI–R , since PostgreSQL does not use SI–C, but a variant of SI–C, for concurrency control. This variant, which we call SI–P (snapshot isolation in ***P***ostgreSQL), is presented in Section 3.3.1. Finally, in Section 3.4 we present SI–PR (concurrency and ***r***eplica control providing SI in ***P***ostgreSQL), our version of the replica control algorithm which is based on SI–P.

3.1 SI-C: Centralized Concurrency Control providing SI

In this section, we present SI–C, an abstract centralized concurrency control algorithm providing SI. Each transaction T_i receives timestamps $TS_i(BOT)$ and $TS_i(EOT)$ at

the begin and end of transaction respectively. A counter is used for this purpose. When a transaction commits the counter is increased and the new value is the transaction's EOT timestamp. When a transaction begins, its BOT timestamp is set to the value of the counter without increasing it. As such $T1$ and $T2$ are concurrent if neither $TS_1(EOT) \leq \mathrm{TS}_2(BOT)$ nor $TS_2(EOT) \leq \mathrm{TS}_1(BOT)$.

In SI–C, each transaction T_i has an *execution phase* and a *check and commit phase*:

1. *Execution Phase*
   - When transaction T_i performs a write operation $w_i(X)$ on data object X for the first time, it checks whether a concurrent transaction T_j that has already committed, also updated X. This is the case, if there is a version of X labeled with T_j and $TS_j(EOT) > TS_i(BOT)$. In this case, T_i aborts. Otherwise, it creates a shadow copy of X that is not visible to other transactions and performs the write operation on the shadow copy. If T_i has already a shadow copy, it performs the update on its shadow copy.
   - When T_i performs a read operation $r_i(X)$ on data object X and T_i has already a shadow copy of X, the read is performed on the shadow copy. Otherwise, T_i reads the last committed version of X as of T_i's BOT timestamp. That is, T_i reads the version of X labeled with T_j such that $TS_j(EOT) \leq \mathrm{TS}_i(BOT)$ and there does not exist another version of X labeled with T_k and $TS_j(EOT) < TS_k(EOT) \leq \mathrm{TS}_i(BOT)$.
2. *Check and Commit Phase:* Upon a commit request, T_i checks whether any concurrent already committed transaction had a conflicting write operation. That is, for each of its shadow copies X, T_i checks whether there is a version of X labeled with T_j and $TS_j(EOT) > TS_i(BOT)$. In this case, T_i aborts and discards all its shadow copies. If there is no conflict on any data object, T_i commits and its shadow copies become visible to other transactions.

In order to provide correctness, there may not be two transactions in the check and commit phase at the same time. The algorithm guarantees SI. Read operations read from a snapshot as of begin of transaction. Out of two concurrent transactions updating the same data object only the one that first enters the check and commit

phase will commit, the other will abort. This is also called the *first-committer-wins* rule. The algorithm performs preliminary checks when it first updates a data object to allow for early conflict detection.

3.2 SI-R: Concurrency and Replica Control providing SI

In [22], the authors propose a suite of different synchronous and update everywhere replica control algorithms depending on the concurrency control provided by the underlying database systems. Among them, one is based on SI. We refer to this algorithm as SI-R. The basic idea of these algorithms is as follows. A transaction T_i can be submitted to any replica. This replica is said to be the *local* replica and transaction 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. At commit time, the writeset containing all write operations is multicasted to all sites using the total order multicast. The delivery order of the writesets will determine the serialization order of the corresponding transactions.

For SI-R it is important to generate globally unique BOT and EOT timestamps and a transaction should have the same timestamps at each replica. For that purpose, the algorithm uses the total order delivery of writesets. When the system starts, each node sets its timestamp counter to 0. Whenever a writeset is delivered, the counter is increased and the transaction whose writeset is delivered, receives the value of the counter as EOT timestamp. Note that, with this, there might be transactions with EOT timestamps that have not yet terminated since the delivery of a writeset happens before the transaction terminates. When a transaction T_i starts, its BOT timestamp is set to a value $TS_i(BOT)$ such that for each transaction T_j with $TS_j(EOT) \leq \mathrm{TS}_i(BOT)$, T_j has already committed.

Each transaction T_i has a *local execution phase* and a *send phase* at the local node, and a *version checking phase*, a *write phase* and a *commit phase* at all nodes. There cannot be two transactions concurrently in the version checking phase. The algorithm

extends SI-C with a locking procedure that takes place in the version checking phase. This guarantees that remote transactions execute conflicting operations in the order of writeset delivery.

1. *Local execution phase:* This phase is exactly the same as in SI-C. During this phase the write operations are gathered to form the writeset.
2. *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 the total order multicast. WS_i contains $BOT(T_i)$.
3. *Lock and version check phase* This phase starts when the writeset is delivered. At most one transaction can be in this phase. T_i receives its EOT timestamp. Then it checks whether any concurrent transaction whose writeset was delivered before T_i had a conflicting update. To do so, T_i does the following for each write operation $w_i(X)$ in WS_i. It checks whether there is a lock on X.
   a.) If not, it checks whether there exists a version of X labeled with T_j and $TS_j(EOT) > TS_i(BOT)$. If this is the case, the lock and version check phase is terminated, and T_i aborts. If T_i was local, it discards all its shadow copies. If there is no such conflicting version of X, T_i receives the lock on X.
   b.) If there is already a lock on X, not only existing versions of X have to be examined but also the future versions created by the transaction that has the lock or transactions that are waiting for the lock. That is, let T_j be the last transaction to modify X before T_i (the last one in the waiting queue for X or the transaction holding the lock if the waiting queue is empty). If $TS_j(EOT) > TS_i(BOT)$, then T_i has to abort (terminating the lock and version check phase and discarding local shadow copies if necessary).
4. *Write Phase:* Whenever a write lock on X is granted and T_i is local, its shadow version becomes a valid version. If T_i is remote, it creates a new version of X, and executes the write operation.
5. *Commit Phase:* When all write locks are granted and operations executed, the transaction commits and releases its locks which are granted to the next one waiting in the queue.

Note that while in the centralized case, check and commit was in one phase,

it is now separated. The reason is that remote transactions have to execute the write operations which can take considerable time. Hence, the algorithm wants to allow non-conflicting remote transactions to execute concurrently while conflicting operations should be executed in the order of writeset delivery. Therefore, locking is performed in an atomic step to enqueue conflicting locks in the correct order.

3.3 SI-P: Concurrency Control in PostgreSQL

PostgreSQL [18] is a well–known open–source DBMS. It implements a multi–version concurrency control mechanism providing SI. However, the algorithm is quite different to SI–C. In this section, we describe PostgreSQL's implementation of SI and refer to it as 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 supports them.

3.3.1 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 tuple X is assigned a unique identifier which is common to all versions of X. Each update creates a new version of X. All versions are kept, even those created by transactions that later abort. We denote a version created by a transaction that committed (aborted) a *committed* (*aborted*) version. An important characteristic is that write operations have to acquire an exclusive lock on the tuple 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, 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

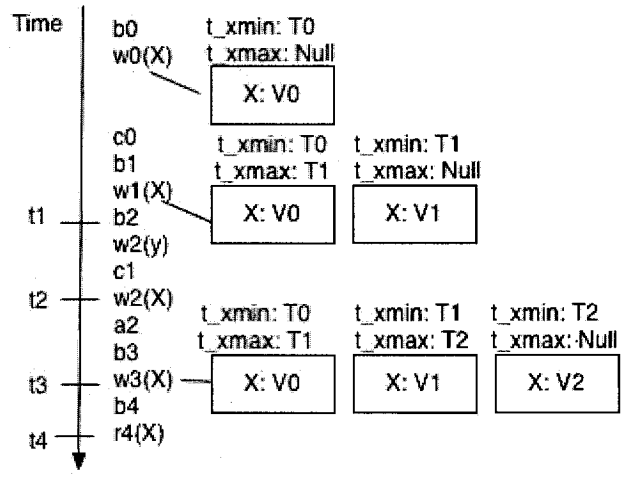

Figure 3.1: Version Creation during Execution

is at most one active version of X in the system. In Figure 3.1 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. Figure 3.1 depicts how transaction $T3$ at time $t3$ takes the version $V1$ created by $T1$ 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 a 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 3.1, $T2$ must abort at time $t2$ because of valid version $V1$ with $t\_xmin = T1$ and $T1$ committed after $T2$ started.

When a transaction T_i performs a read operation on X, 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 (1) either NULL, (2) refer to an aborted transaction, or (3) 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. In Figure 3.1 when $T4$ performs the read at $t4$ it reads $V1$ ($T1$ committed before $T4$ started), and not the active version $V2$.

Determining the snapshot In order to determine valid and visible tuples, a transaction must know which transactions committed, aborted, or were active when it started. PostgreSQL keeps information about each active transaction in shared memory. Part of the information of transaction T_i is a `snapshot` struct that is created when T_i starts. `snapshot` contains $xmax$, 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 $Ti_{concurrent} = \{TID \in xip \vee TID \geq xmax\}$ 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 $Ti_{committed} = \{TID | TID \notin T_{concurrent}$ and T_{TID} committed$\}$. A transaction requires a fast mechanism to determine whether a terminated transaction has aborted or committed. For that purpose, PostgreSQL keeps a file called *clog* whose tail is buffered in shared main memory[1]. Whenever a transaction terminates, a log entry will be inserted into *clog* to record the commit or abort outcome. *clog* provides a fast access method to determine the outcome of a transaction given its TID. As an example, when $T2$ starts in Figure 3.1 at $t1$, xip is $\{TID_1\}$, $xmax = TID_3$, and *clog* contains a commit entry for $T0$.

The protocol Each transaction T_i has two phases

[1] *clog* is also used for recovery purposes.

1. *Execution Phase*
   - When transaction T_i performs a write operation $w_i(X)$ on data object X for the first time
     - *Check:* It first checks if there exists a version of X such that $t\_xmin \in Ti_{concurrent}$ and the transaction with $TID = t\_xmin$ has already committed. This is the same check as in SI-C.
     - If such version exists, then T_i is aborted.
     - If no such version exists, T_i requests an exclusive lock for X.
     - If the lock is granted immediately, T_i looks for its visible version of X, that is the version V of X such that $t\_xmin \in Ti_{committed}$ and $t\_xmax \notin Ti_{committed}$. Note that because T_i passed the version check and has a lock on X, t_xmax is actually either $NULL$ or the TID of an aborted transaction. It cannot be the TID of an active transaction or a committed transaction that was concurrent to T_i. T_i makes a copy V' of V, and sets t_xmax of the old copy V and t_xmin of the new copy V' to its own TID_i.
     - 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 at the step above labeled *Check*.
   - When transaction T_i performs a successive write $w_i(X)$ on a data object, it already holds the lock on X and has created a new version V'. It performs the update simply on V'.
   - When transaction T_i performs a read operation $r_i(X)$ on data object X, it reads its own version V' if a previous write on the same object was performed, or it reads the version V of X such that $t\_xmin \in Ti_{committed}$ and $t\_xmax \notin Ti_{committed}$. Note that in this case t_xmax might contain the TID of a concurrent transaction T_j (meaning that T_j has performed an update and created a new version, or deleted the tuple), but V is still visible to T_i because T_i ignores T_j's updates.
2. *Termination Phase.* Upon the commit request or abort request for T_i, T_i updates

clog, releases all locks, and wakes up all transactions waiting for one of these locks.

The difference of SI–P compared to SI–C is that there is a locking procedure in the execution phase. When a transaction wants to create a new version of a data object, it tries to lock that object after the version check. If there is no lock on the object, the lock will be granted. Else, the lock request will be enqueued. By this locking procedure, conflicting write operations are serialized. When a transaction T_i holding the lock commits and wakes up a waiting transaction T_j, T_j performs again the version check. This time, T_i's version exists leading to the abort of T_j. If T_i aborted, it also wakes up T_j. In this case T_j's check will succeed, and it will again attempt to get the lock. If more than one transaction is waiting, all are woken up, all perform the check, and either abort, or compete again for the lock. It is a neat approach which takes advantage of the fact that version checks are very fast due to the `snapshot` struct and the *clog*.

3.4 SI-PR: Concurrency and Replica Control providing SI based on SI-P

By analyzing SI–P it becomes clear that the SI–R algorithm of Section 3.2 must be adjusted. The first major issue are global timestamps. $TIDs$ are local at each replica and there is no guarantee that a transaction can get the same $TIDs$ at all sites (because each site will have different read-only transactions that are not sent to other sites). Hence, we use as in SI–R the delivery order of writesets to generate globally unique timestamps. However, the internal system of PostgreSQL works with $TIDs$ and we do not want to change this system because this would have an effect on many different modules of PostgreSQL. As such, each transaction keeps locally its TID, and at the same time, each update transaction will receive a global identifier, referred to as GID, which will be the same at all sites, and is determined by the order the writesets are delivered. The replication component at each site keeps an internal table that allows for a fast matching between GID and TID of a transaction. One

important thing to keep in mind is that for a local transaction, a TID is generated at the start of the transaction, the corresponding GID is only determined when the writeset is delivered at commit time. For a remote transaction, however, GID and TID are generated at the same time when the writeset is delivered.

Another major change we did is that we do not allow remote transactions to execute concurrently but they execute serially one after the other. In particular, whenever a writeset is delivered for either local or remote transaction, the transaction has to completely terminate (commit or abort) before the next writeset is delivered. This is fast for local transactions. However, for remote transactions it means they go through the checking phase and the execution phase before a new writeset is delivered. The reason is that concurrent execution requires some locking as explained in Section 3.2. However, the centralized version of PostgreSQL uses its own locking mechanism, to detect write/write conflicts during execution. These locks behave quite differently than the locking scheme of SI–R. Since we did not want to change the locking scheme of PostgreSQL for modularity reasons, we decided to run remote transactions serially. Section 7.1 outlines how our algorithm could be optimized to allow concurrent execution of non–conflicting remote transactions.

We distinguish between the execution of a local transaction and a remote transaction.

- *Local Transaction*
  1. *Execution Phase:* The execution phase is the same as in the centralized concurrency control protocol in PostgreSQL, SI–P, with some additional steps. Purely TIDs are used for conflict check, and the same locking procedure for writes is used as in SI–P. For each write operation $w_i(X)$ on tuple X, T_i retrieves the version V of the X created by T_j such that T_j is the last transaction that updated the tuple and committed before T_i started. T_i makes its own copy V' of V and performs the update on it. At the same time, it retrieves the GID of T_j (which already exists since T_j already committed). Both the new version V' of the tuple and the GID of T_j are added to the writeset.
  2. *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 the total order multicast.

3. *Commit Phase:* Upon delivery of the writeset, the GID is determined and added to the internal table together with TID. The rest is the same as the termination phase in SI–P.

- Remote Transaction: Upon delivery of the writeset for remote transaction T_i, the GID is determined, a transaction is started locally, and both GID and TID_i are added to the internal table.
  1. *Version Check and Early Execution*:
     a.) For each tuple version V' of X in WS_i, the version check is performed. The idea is the following: T_i retrieves the local valid version V of X according to its local TID_i. That is, it retrieves the version V created by a transaction T_j that was the last to commit and update X before T_i started locally on this remote site ($t\_xmin \in Ti_{committed}$ and $t\_xmax \notin Ti_{committed}$ according to its local `snapshot`). Now it retrieves the GID of T_j and compares it to the GID attached to V' in WS_i.

     b.) If the GIDs are different, a concurrent transaction had updated X and committed before T_i (it had committed before T_i started on this remote site, but it was concurrent to the execution of T_i on its local site). Hence, the check fails, and T_i aborts.

     c.) If the GIDs are the same, the last version of X is still the same as when T_i executed at its local site. Hence, the version check succeeds. In this case, T_i requests an exclusive lock for X.

     d.) If the lock is granted immediately, T_i sets t_xmax of V and t_xmin of V' to its own TID_i.

     e.) If there is already a lock on X, this lock belongs to an active local transaction T_j which either has not yet send its writeset or whose writeset has not yet been delivered (there is only at most one remote transaction active; and local transactions whose writesets had been delivered are guaranteed terminated). In principle, T_j should be aborted because T_i's writeset is delivered before T_j's writeset, and hence, T_i should be serialized before

T_j. However, we do not abort it immediately but delay further actions on X until T_i has passed the version checks on ALL tuples in WS_i.

2. *Late Execution:* If the transaction has not yet aborted, it has passed the version checks of all tuples in WS_i. Furthermore, it has performed the updates on tuples for which no local active transaction had a lock. 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. These transactions have to be aborted. Hence, for each such tuple, T_i requests again a lock, and if the local transaction still holds it (it might have aborted in between for some reason), T_i sends an abort request to T_j. When T_j receives this request, it aborts, releases the lock which is directly granted to T_i (different to SI-P where all waiting transactions are woken up). T_i updates V and V' of X.
3. *Commit Phase:* After all tuples have been updated, T_i updates *clog* and releases all locks, waking up all waiting transactions.

3.5 Delay Abort Local Transactions

It is very important not to abort a local transaction prematurely. 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. Figure 3.2 provides an illustration: Assume three nodes N1, N2 and N3. Assume that so far there was one transaction with GID=1 updating X and Y at all sites. Now N1 has local transaction T_1 updating X, N2 has local transaction T_2 updating X and Y, and N3 has local transaction T_3 updating Y. All three send their writesets. Assume the delivery order is T_3 before T_2 before T_1. On N1, T_3 gets $GID = 2$, passes the check (GID=1 both for the visible version V and the version V' in the writeset), executes and commits. Then T_2 starts the check, finds local transaction T_1 holding lock on X and T_1's writeset has not yet been delivered. Assume now that T_2 signals T_1 to abort, then checks on Y. It retrieves T_3's version of Y as the visible version with $GID = 2$. It detects the conflict (GID should have been 1) and aborts. This is, on N1, both T_1 and T_2 abort. However on N2, when T_3's writeset is delivered, T_3 aborts T_2 because of the conflict on Y

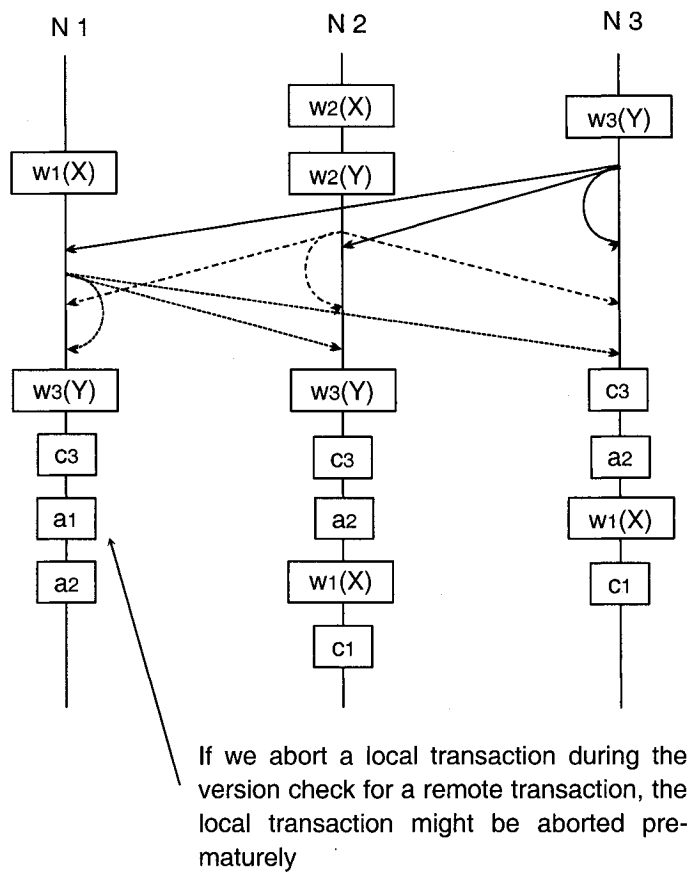

Figure 3.2: Example of Premature Abort

and T_3 has to be serialized before T_2. When now T_1 arrives, T_1 passes the validation test and commits. On N3, upon delivery of T_3's writeset, T_3 commits. When T_2 is delivered, the conflict is detected (visible version has $GID = 2$ instead of 1) and T_2 is aborted. When the writeset of T_1 is delivered, T_1 will commit. In summary, the abort of T_1 on N1 was wrong because T_2 triggers the abort and then aborts itself. Hence, we have to delay aborts to local transactions until we are sure that the remote transaction causing the abort will commit.

However, we do not delay all execution. During the checking phase of a remote transaction, if there is no conflict with a local transaction, we immediately perform the update. That is, we do not completely separate the version check from the execution.

Accessing tuples is expensive because they have to be found and possibly loaded from disk. If we completely separate checking phase and execution phase, each tuple has to be loaded twice, which can be expensive. By combining both phases whenever there is no conflict we speed up execution.

At this point we like to explain why 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 which is serialized before T_i, T_i would have been aborted by T_j in T_i'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.

Chapter 4

PostgreSQL-R Project

PostgreSQL is an open source, object–relational database management system [26, 28]. It has a long history starting at the University of California at Berkeley in 1986 when Prof. Michael Stonebraker began the project, originally called Postgres. The main goal of this project was to show that a relational database system could cope with modern demands of extensibility. It was called an object–relational database because it was a relational database system integrated with some object–oriented features, such as inheritance, user–defined data types, operators, and functions. Now this project is continuously developed by a group of people in the open–source community under the name of PostgreSQL. In addition to those object–oriented features, PostgreSQL also has other core features in the latest version. For instance, it supports SQL92 and SQL99. It supports internal procedural languages, including a native language called PL/pgSQL. This language is comparable to Oracle's procedural language, PL/SQL. It provides several APIs, e.g., ODBC, C, C++, JDBC, Perl, Tcl/Tk and Python. Its internal services have been largely improved, providing more efficient buffer management, write-ahead-logging, and, as described in Section 3.3., a multi–version concurrency control mechanism.

The Postgres–R project extends PostgreSQL. The project was initiated by Bettina Kemme, Gustavo Alonso, Win Bausch, and others at ETH Zürich. The goal of the project is to provide synchronous database replication strategies using GCS. In [5], Win Bausch implemented an update everywhere replica control algorithm from [22]

based on strict 2PL. This implementation laid out the basic architecture of Postgres–R. It also provided general functions to generate writesets and apply them efficiently at remote sites without re–executing the SQL statements. This first version was based on PostgreSQL 6.4 which used locking as its concurrency control method. The PostgreSQL community took the version implemented by Win Bausch (with several changes done by Bettina Kemme) and migrated it to PostgreSQL 7.2. However, the developed version was never correct, since PostgreSQL 7.2 uses the SI–P algorithm described in Section 3.3 and no more 2PL. Mabrouk Chouk [10] took this modified version and transformed it to a master–slave replica control protocol (which did not need to consider concurrency control issues) and added recovery. For my thesis, I have taken Mabrouk Chouk's version and changed it again back to update everywhere by developing and implementing the SI–PR algorithm of Section 3.4. The main architecture and the basic writeset functionalities, however, are still the same as in the original Postgres–R by Win Bausch.

In this chapter, we will first describe PostgreSQL and its components, and then describe the main architecture and functionalities of Postgres-R as existing at the begin of this thesis work.

4.1 PostgreSQL

4.1.1 General Architecture

PostgreSQL uses a process–based client/server architecture [26, 28]. A PostgreSQL session consists of the following cooperating processes: A supervisory daemon process (the *postmaster*), the user's *frontend* application or command execution environment (psql), and one or more *backend* (Postgres) processes which are the only processes directly working on the tuples. The postmaster process is always running, waiting for requests from frontends. Frontend applications that wish to access a given database make calls to the client library (libpq/ODBC). The library sends user requests to the postmaster, which starts a new backend process and connects the frontend process to that new backend. From that point, the frontend and the backend communicate

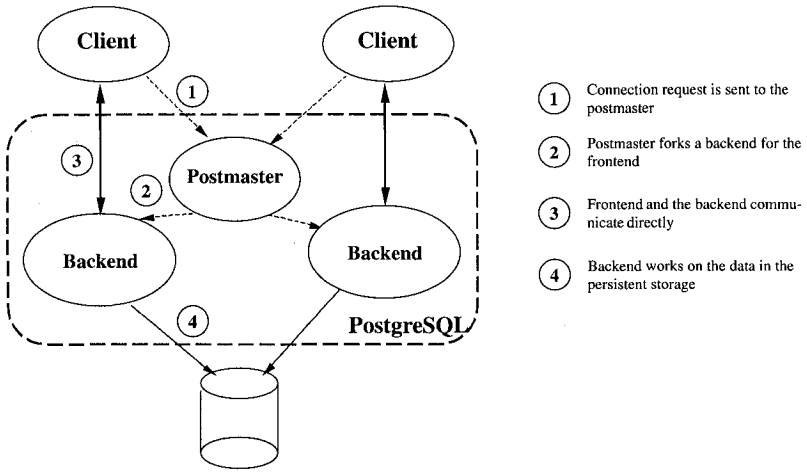

Figure 4.1: Architecture of PostgreSQL

without intervention by the postmaster. The backend executes commands on behalf of the application program. A simple request–answer communication model is used between the application and the backend. When the client disconnects, the connection is dropped and the backend exits. The message communication between client applications and the server can be configured using TCP socket or UNIX socket.

Since PostgreSQL is a process–based system, only one transaction can be executed in a backend at a time. Each backend has a `Proc` struct in the shared memory. The `Proc` struct stores the state of a process. It contains the information about the process, e.g. process ID, TID and the link to the `Proc` struct for the next process in the shared memory queue. It also stores the data related to the transaction currently executing in the backend, e.g. transaction ID, execution state, locking information etc.

In a backend, the *main* routine is the gateway connecting database and the client application. When a backend starts, it first checks the options handed over by the postmaster, e.g. the size of the buffer pool, whether the client is allowed to modify system tables etc. The backend process then initializes the underlying data structures based on these options. Before it starts to receive any query request from the client, the backend has to register signal handlers to handle certain asynchronous events,

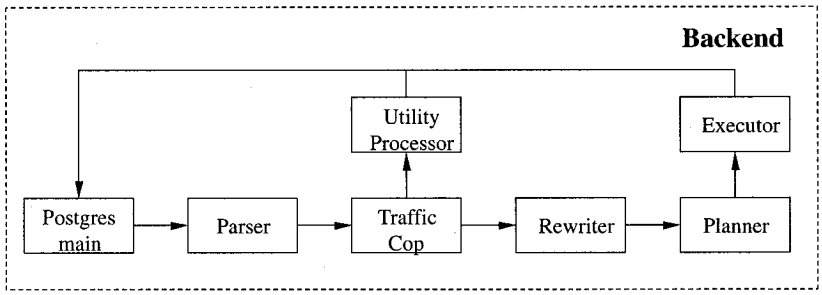

Figure 4.2: Query Processing Modules

e.g. cancel current query request, deadlock detection request, exit request etc. The signal mask is also set to determine which signals can be raised. This is part of the standard exception handling mechanism. Signal handlers to handle these different signals are provided. Now the backend is ready to enter the query loop. For each client query, the loop is executed once.

Queries submitted by the client are first passed to the *parser*. The parser checks the syntax and generates a *parse tree* which will be handed over to the *traffic cop*. The traffic cop categorizes the queries into two types: *complex queries* or *utility queries*. The complex queries in PostgreSQL are UPDATE, SELECT, INSERT and DELETE. These queries are sent to the next stage, the *rewriter*. The utility queries are directly sent to the *utility processor*. The rewriter re–writes the parse tree of complex queries to an alternative form by applying any applicable rule. By this module, PostgreSQL supports a powerful rule system for the specification of views. The next module is the *planner*. The planner generates various execution plans and makes a cost–estimate–based selection to get the optimal plan for further processing. Finally, the plan will be processed by the *executor*. If a transaction aborts at any time point, an abort routine is executed. Then the backend jumps back to the beginning of the query loop.

4.1.2 IPC

There are three *Inter–Process Communication (IPC)* mechanisms used in PostgreSQL: *sockets*, *UNIX signals* and *shared memory*. The communication between the frontend process and the postmaster, and between the frontend and the corresponding backend is done through sockets. The postmaster and the backends can communicate with each other using UNIX signals in some unusual circumstances, e.g. shutdown. The communication between backends basically relies on shared memory.

The socket port of the postmaster is predefined and assumed to be known by clients. When a client sends a request to the postmaster, a new port will be assigned to the client. Then, the socket connection between the client application and the backend is created and the two processes can exchange messages. Each message has a message header and a sequence of bytes as message body. The message might be sent in several rounds due to the limitation of the packet size.

Signaling is another basic IPC mechanism. When a client sends a disconnection message abnormally, the postmaster sends a signal to the associated backend. When the postmaster exits itself, it notifies every backend to abort by sending a signal. The backend also can send a signal to the postmaster notifying that a table is full, a password has changed etc. There is no signaling mechanism used in the communication between the backends. However, it is definitely an option.

The shared memory management is more complicated than managing the previous two IPC mechanisms. PostgreSQL implements a shared memory management module inside the *storage management subsystem* which allows to easily create an object within the shared memory. Objects can be accessed by all processes who have a pointer to the object. When the postmaster starts a new backend, it allocates shared memory for it. As we have mentioned, each backend process has a `Proc` header which contains the link to the next process in a shared memory queue. Lock tables are also in shared memory. To locate an item in shared memory, the process has to know the start of the shared memory region and the offset relative to the starting pointer. Since the shared memory is shared, conflicting operations must be synchronized to ensure data consistency. Lightweight locks are used for this purpose.

4.1.3 Locking Mechanism

The *lock management module* is another component of the storage management subsystem. It keeps track of requests for locks and grants locks. This module, together with the *transaction management subsystem*, provides concurrency control in the system. There are three types of locks in PostgreSQL:

- *Spinlock* is a very short–term lock. It is primarily used as infrastructure for lightweight locks. Busy–loop and timeout mechanism are used in the implementation. There is no deadlock detection and automatic error handling for this type of lock.
- *Lightweight lock (LWLock)* is the type of lock which is typically used to lock access to the data structures in shared memory. It has exclusive and shared lock modes. There is no deadlock detection, since it is assumed that a lock is released before a new one is requested. But the system automatically releases locks upon an error. LWLocks are implemented using semaphores. A process waiting for a LWLock is blocked and locks are granted in arrival order.
- *Regular locks* support a variety of lock modes. Deadlock detection is provided and locks are automatically released at transaction end.

One important note about spinlocks and LWLocks is that these two types of locks, unlike regular locks, hold off interrupts until all such locks are released.

Locks are the building block of the concurrency control mechanism. There are eight lock modes which describe the type of the lock, e.g. AccessShareLock, ShareLock and ExclusiveLock etc. The fundamental data structures in the lock management model are the `LOCK` struct and the `HOLDER` struct.

- Every lockable object has a `LOCK` struct whenever at least one lock is granted on the object. It contains:
  - *locktag*, a unique identifier of the lockable object. It is used for hashing locks in a shared memory hash table.
  - *grantMask*, bitmask for lock types already granted
  - *waitMask*, bitmask for lock types waiting to be granted
  - *lockHolders*, queue of `HOLDER` objects associated with the lock

- *waitProcs*, queue of processes waiting on the lock

- Each transaction that is holding or requesting a lock on a `LOCK` struct has a HOLDER struct. It contains:
  - *holdertag*, unique identifier of the holder. It is the hash key for the holder hash table in shared memory.
  - *lockLink*, link to all the `HOLDER` structs of the transaction
  - *procLink*, link to all the processes which hold the lock

When a transaction tries to request a lock on an object, it creates a locktag and a holdertag which are used to search the `LOCK` and `HOLDER` structs in the lock hash table and the holder hash table. If no corresponding `LOCK` or `HOLDER` struct could be found, a new one is created. Then, if there is no lock granted or the transaction has already a lock on the object, the lock is granted. Otherwise, a conflict check is necessary. The type of lock requested is checked against the grantMask as well as the waitMask in the `LOCK` struct. If there is no conflict, the lock is granted and the transaction is allowed to perform the operation on the data object. Else, the backend process executing the transaction is blocked until a holder releases the lock. It implies that PostgreSQL does not support a process to request multiple locks at the same time.

When a process joins the waiting queue, it is normally appended to the end of the queue. However, if the process already holds other locks that conflict with the request of some previous waiter, it puts itself in the queue just in front of the first such waiter. To synchronize access to lock related data, a masterlock (LWLock) is acquired before the Lock struct data is accessed. Upon releasing a lock, all waiting processes are woken up in the order in which they are waiting in the waiting queue. However, 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.

If a process has to wait for a lock, it sets a timer and goes to sleep. This timer is used to wake up the process after a certain sleep time to perform deadlock detection. The standard directed–graph (wait–for graph) method is used in the deadlock detection mechanism in PostgreSQL. If a deadlock is detected, the process will abort. When a process appends itself to the waiting queue, it checks for *two–transaction*

deadlocks (deadlocks involving only two transactions) with each waiter in the waiting queue. This step avoids launching the deadlock detection routine later, since the deadlock detection is very expensive.

4.1.4 Transaction Abort Mechanism

In PostgreSQL, there are four scenarios in which a transaction might abort. One scenario happens when an exception is raised. In this case, the abort routine is called to abort the current transaction and the backend starts again at the query loop. The second scenario is that a transaction fails the version check within the concurrency control component. The third one takes place when the client sends an ABORT utility command. In the latter two cases, the backend is in a safe state. Safe means that the transaction can abort immediately, without messing up the system. They are the normal query abort scenarios. The abort routine is called directly and the backend jumps to the beginning of the query loop. The fourth and last scenario is a bit tricky. It happens when the backend receives a query–cancel signal (the client has sent a cancel connection request) or shutdown signal from the postmaster. When the backend receives such signal, it can be in any state or in the middle of any operation. There is no guarantee that the database would remain consistent if interrupts are allowed at an arbitrary point in execution. Therefore, PostgreSQL declares three volatile variables: $ImmediateInterruptOK$, $InterruptHoldoffCount$ and $CritSectionCount$.

1. ImmediateInterruptOK is set to 1 if the backend is waiting for input or a lock. At these two spots, interrupts are allowed.
2. InterruptHoldoffCount is incremented when low–level subroutines manipulate data structures in shared memory. Only if InterruptHoldoffCount = 0, the signal can be processed.
3. CritSectionCount is incremented when the backend performs an operation on the Write Ahead Log. An exception will force a system–wide reset if CritSection-Count is not zero at the time of the exception. Therefore, interrupts should not be allowed.

Only if the conditions set by these three variables are satisfied, the signal can be processed right away. If it is not 'safe', a flag, QueryCancelPending, is set. During query processing, the backend checks QueryCancelPending at some 'safe' spots and processes the signal. The signal is treated as an exception, and the control will be redirected to the transaction abort routine.

4.1.5 System Catalog

Database metadata is "data about data". It describes the content of the database, e.g. information about tables, attributes, indexes etc. In addition, it also contains functions that manipulate relations. PostgreSQL provides *system catalogs* to store such metadata. There exists a catalog containing information about all tables, a catalog containing information about attributes etc. Catalogs by themselves have table structure. All of the catalogs are maintained and accessed via the catalog subsystem. The catalogs are cached. And there is an efficient way to find a record in a catalog from the system cache by index lookup, without going through the execution path to be used for queries on regular tables.

4.2 Postgres-R

4.2.1 Architecture

The architecture of Postgres–R is depicted in Figure 4.3. Postgres–R extends PostgreSQL with three new components: *remote backends*, *replication manager* and *communication manager*. The original backends are now called *local backends*. They execute local transactions that are submitted by clients connected to the particular Postgres–R instance. A remote backend is a variant of the original backends. It processes the writesets propagated from other nodes in the system. The communication manager is in charge of the message exchange between the replication manager and the GCS hides the details of GCS. The replication manager is responsible for starting remote backends, and coordinating the execution of backends and their communication with the communication manager. In addition to these new components, the

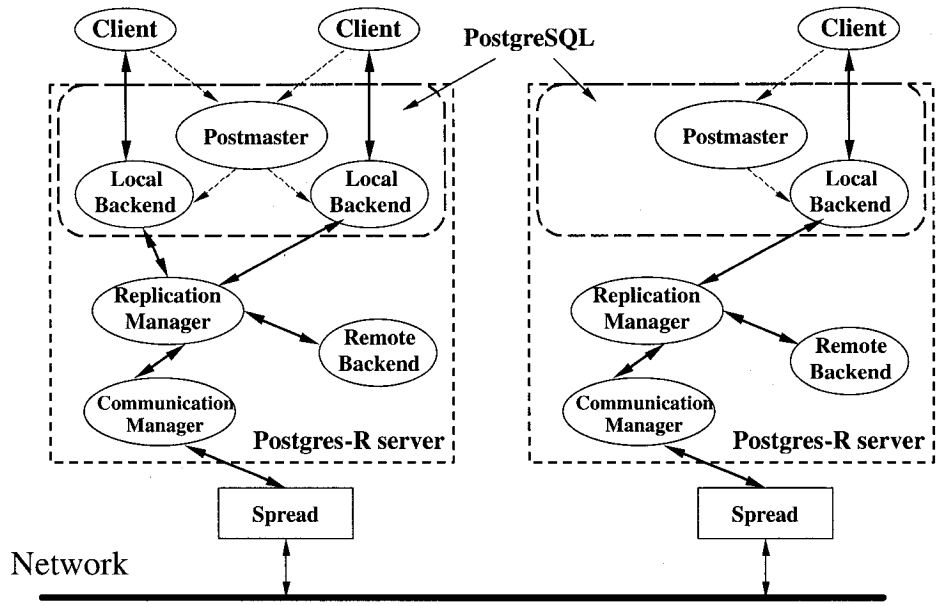

Figure 4.3: Architecture of Postgres-R

struct `writeset` is used to bundle all the write operations of a transaction into a single message.

When a database server starts, the postmaster forks the replication manager process. Then, the replication manager starts the communication manager and a remote backend. The replication manager listens on two main sockets: the replication manager server socket and the group communication socket. When a local backend wants to send a writeset for the first time, it connects to the replication manager through the replication manager server socket. A new dedicated `connection` struct and a new socket are created for this backend. The `connection` struct will be kept in a `connection list` until this local backend is closed. The remote backend has its own socket and `connection` struct which is also kept in the `connection list`. The `connection` struct contains information about the associated backend: host ID, process ID, backend type, TID of current transaction, state of the backend and the pointer to the socket. The state of the backend is used to identify which stage the backend is in, as seen by the replication manager. Each message exchanged between backends and replication manager has a header containing message type, message size

and other information.

There exist four procedures which are used to handle messages: one for each communication direction (from backend to GCS or from GCS to backend) and backend type (remote or local). The main routine of the replication manager waits for messages on all of the enabled sockets. When the main routine observes a message and receives the message entirely (based on the message size), it passes the message to the according message handling routine.

4.2.2 Writesets and Their Application on Remote Nodes

A Writeset carries information about the write operations of a transaction. *Statement-level* replication is sending the original query in plain text to the remote sites which will re–execute the query. With *tuple-level* replication, the modified tuples are sent. At the remote site, these tuples are directly accessed and updated. The advantage of tuple-level replication is that there is no hidden dependency. For instance, using statement–level replication, if an update fires a trigger that again contains update statements, there is no guarantee that the triggered update will be executed on each replica. Furthermore applying a tuple–level replicated writeset at a remote node is usually faster than re–executing SQL statements. In our current implementation, tuple-level replication is used for complex queries. Utility commands use statement-level replication. In the writeset, the data is grouped by relations. For each relation, there exists a list of `query` structs, one for each query on this relation. With statement-level replication, the `query` struct contains simply the query text. With a tuple-level replication, it contains a `tuple collection`. For each changed tuple there is a collection of modified attribute values, their attribute numbers in the relations (PostgreSQL identifies attributes internally not by their name but by an assigned number), and possibly the primary key values of the tuple (for DELETE and UPDATE). Hence, the `tuple collection` can be seen as a two–dimensional array, the first dimension for the tuples and the second for the attributes.

When processing the writeset, the remote backend processes the queries following the order in which they appear in the writeset. For statement-level replication, the

execution path is the same as it is for a local transaction (parser, planner, executor). For each tuple to be modified in tuple-level replication, the remote backend first retrieves the tuple from the database in two steps. First, the `tuple descriptor` is retrieved based on the relation name. The `tuple descriptor` contains all the information about a tuple in the relation, e.g. attribute names and their order numbers and which attributes constitute the primary key. This retrieval is fast since the information is stored in the catalog. Second, the index on the primary key, which is created automatically when the table is created, is then used to efficiently find and retrieve the tuple. Now the remote backend can directly perform the change on the tuple (skipping most of the normal query execution steps).

4.3 Distributed Recovery

Although recovery is beyond the topic of this thesis, we want to briefly talk about the recovery mechanism in the master/slave Postgres-R. In PostgreSQL, there is a centralized recovery mechanism, which uses a *Write–Ahead–Log (WAL)*. For each modified tuple, the modified attribute values and their attribute numbers within the relation are logged. Before any change is written into the database, it must be logged into WAL first. At commit time, the log is flushed to disk to guarantee durability. The postmaster periodically performs checkpointing (flushing all dirty pages to disk). When the database starts after a crash, the postmaster will check the transactions which are committed after the last checkpoint in the WAL and redo any transaction which is logged in WAL but whose updates were not written to the database before the crash. Those aborted transactions in the WAL are ignored.

In [10], Mabrouk Chouk implemented a distributed recovery protocol for the master–slave approach in Postgres–R. When a node restarts after a crash, it first has to perform centralized recovery. After that, it has to synchronize with other nodes. During the downtime, the other nodes might have executed many update transactions. The recovering node has to get the changes performed by these missed transactions from a peer node. For that, it has to identify the missed transactions. Since local TID's are different on different nodes, Mabrouk Chouk introduced *Global*

Transaction Identification, GID, that identifies a transaction throughout the system. The replication manager keeps a distributed recovery log that records for each transaction its local TID, the GID and a pointer to the WAL containing the transaction's updates. During recovery, the recovering node receives from the peer node all transactions that appear in the peer node's distributed recovery log but not in recovering node's log, and applies them. Of course, the recovering process has to coordinate with the concurrent processing of new transactions. Although the new update everywhere version of Postgres–R does not yet support recovery, we kept the necessary data structures, so that the system can easily be extended to provide such functionality.

Chapter 5

Implementation Details

5.1 Overview

While the main architecture of Postgres–R has remained the same, the integration of SI–PR changed the execution of a transaction considerably. We split the execution of transactions into different steps, possibly performed by different components. The start of a new step is triggered by events, e.g. the reception of a request or message. Events related to replica control protocol, their ordering and interaction between replication components are described in Figure 5.1 together with Figure 5.2 and Figure 5.3.

First, we depict the execution of a local transaction. Figures 5.1 and 5.2 help to understand the interaction:

1. A transaction starts when a client submits a command to a local backend. The transaction boundary can be explicitly issued by the client using BEGIN and END/ROLLBACK commands. If the client does not use these statements, PostgreSQL will execute each individual SQL statement as a transaction. When a query arrives, the local backend enters the Execution Phase. During query execution, if there is any update performed, it is added to the writeset. As indicated in Section 3.4, the GID of the transaction whose version was copied is included. (Step 1 in Figure 5.1)
2. Upon receiving a commit request, the local backend checks the writeset. If the

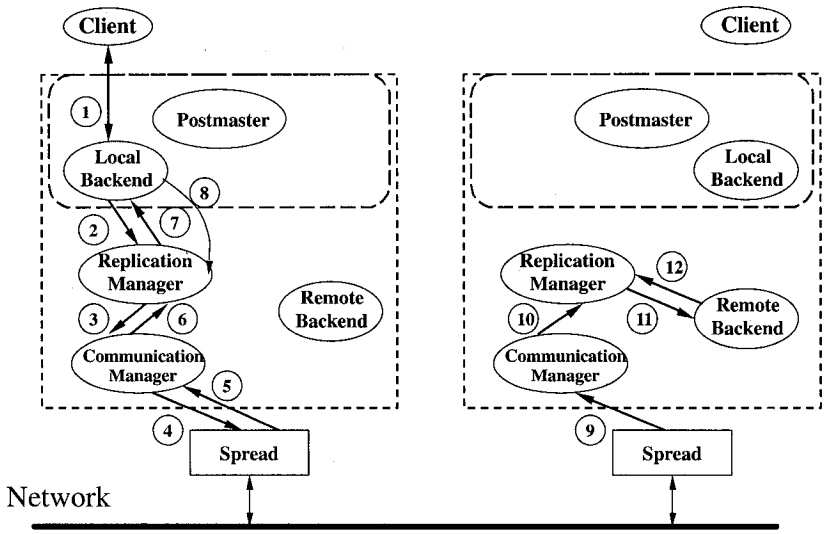

Figure 5.1: Replication Events in Postgres-R

writeset is empty, the transaction commits and results are returned to the client. If there is anything in the writeset, it will be sent to the replication manager. (Step 2 in Figure 5.1)

3. When the replication manager receives a writeset from the local backend, it forwards the writeset to the communication manager. (Step 3 in Figure 5.1)
4. The communication manager uses the total order multicast service, provided by the GCS, to multicast the updates to all of the nodes in the system, including itself. Upon receiving a message from the GCS, the message is forwarded to the replication manager. (Steps 4, 5 and 6 in Figure 5.1)
5. Upon delivery of a writeset, a GID is assigned to the corresponding transaction (details in Section 5.2). When the replication manager finds out that the writeset received is for a local backend, it sends a RECEIVED notice to the local backend together with the GID for the transaction. To guarantee serializability, the channel from the communication manager is blocked. (Step 7 in Figure 5.1)
6. When the local backend gets the delivery notice, it records the GID, commits the transaction and sends CHECKED to the replication manager. (Step 8 in Figure 5.1) Upon receiving the acknowledgement from the local backend, the

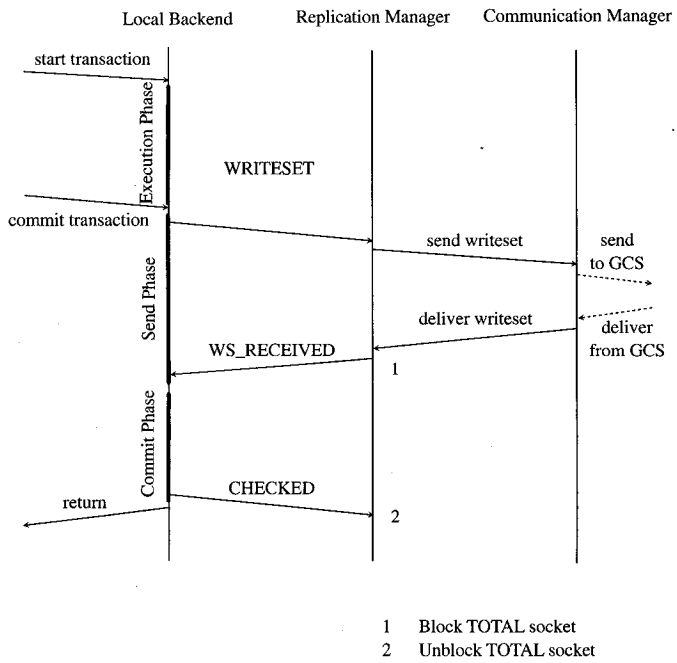

Figure 5.2: Interactive Diagram for Local Transaction

replication manager re–opens the channel from the communication manager to receive the next writeset.

The execution of a remote transactions is shown in Figure 5.1 and 5.3:

1. When the communication manager receives a writeset delivered by the GCS, it forwards the writeset to the replication manager. (Step 9, 10 in Figure 5.1)
2. When the replication manager receives a writeset, it assigns GID to this remote transaction and sends the writeset, together with the GID, to the remote backend. The replication manager blocks the channel from the communication manager. (Step 11 in Figure 5.1)
3. The remote backend records the GID, and performs the version check. If it passes the version check for all tuples, the write operations in the writeset are processed (aborting local transactions if necessary). Then it sends a READY message to the replication manager. (Step 12 in Figure 5.1)

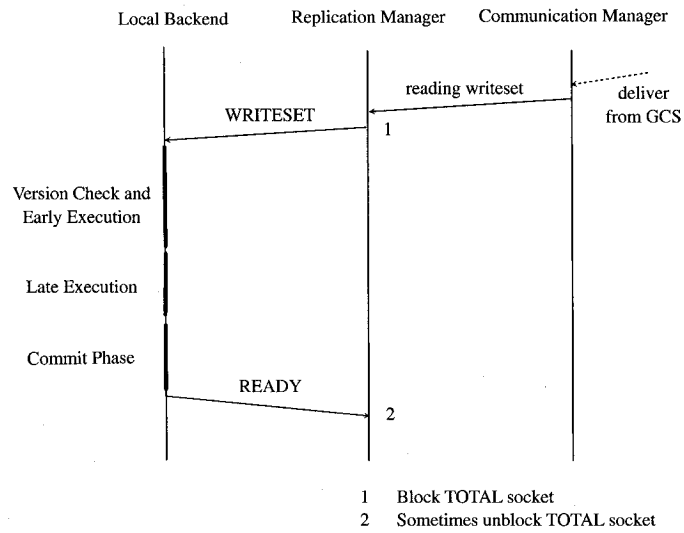

Figure 5.3: Interactive Diagram for Remote Transaction

4. Upon receiving a READY message, the replication manager will enable the channel from the communication manager once all local transactions, that had to be aborted due to conflict with the remote transaction, have completed the abort (see Section 5.2.2 for more details).

So far, we have described the successful execution of transactions. However, transaction abort might happen at certain states of the execution. We will discuss this in Section 5.2.1.

In summary, the major changes we did on Postgres–R to implement SI–PR have been:

- The control flow within the replication manager has been modified to follow the steps depicted above.
- A GID generation mechanism has been implemented.
- A new system catalog which maintains the GID and the corresponding TID has been added to the system.
- The abort mechanism has been enhanced to allow remote transactions to abort conflicting local transactions.
- A version check mechanism for remote transactions has been implemented. It

Message Type	Description
MSG_OPENING	A local backend sends a connection request to the replication manager
MSG_CLOSING	A local backend sends a disconnection request to the replication manager
MSG_WRITE	A writeset message. It can be send from a local backend to the replication manager, from the replcation manager the communication manager or from the communication manager
MSG_WS_RECEIVED	The replication manager sends writeset delivery notice to a local backend
MSG_CHECKED	A local backend sends a commit notice to the replication manager
MSG_READY	A remote backend sends this message to notify the replication manager that it has finished processing a writeset
MSG_ABORT	A local backend or a remote backend notify the replication manager about its abort

Table 5.1: Messages between the Backends and the Replication Manager

requires to match local TIDs with GID.

In Table 5.1, we summarize the message types that exchange between the backends and the replication manager.

5.2 Replication Manager

The replication manager is one of the major components in Postgres–R. It is the coordinator of the replica control protocol on each node. One of the important roles of the replication manager is to create the GIDs. The GID should be unique within the whole replicated system, since it takes the role of the EOT timestamp in the concurrency control algorithm. Furthermore, each node must give a transaction the same GID (while it can have different local TIDs). The replication managers keeps a GID counter. When the replication manager delivers a writeset, it increments the GID by one and assigns the new value to the transaction associated with the writeset. If a transaction aborts but its writeset had been delivered, the GID counter will be decremented by one. Using this approach, we have not only unique but also continuous GIDs. This continuity can significantly simplify the distributed recovery

protocol (discussed in chapter 7).

At startup of a node, the GID counter must be initialized accordingly. When the node joins a running system, it has to perform distributed recovery (Section 4.3). In this case, the peer node can give the current GID. In a newly starting system, the GID can be set to zero. At a restart after a total failure, the GID can be retrieved from a special catalog (see Section 5.3.2).

5.2.1 State Machines

There are two state machines, one for local transactions and one for remote transactions, which show the state of the transaction as seen by the replication manager. These state machines specify the framework of our replication protocol. Note that these two state machines depict the protocol from the replication manager point of view. Later, we will describe state machines as observed by the backends. The difference between the state machines in the replication manager and in the backends is due to time delay between sending and receiving a message. We must pay attention to this delay in our implementation. We describe the local state machine (Figure 5.4) as follows. Before a local backend sends its first writeset, the replication manager does not know about its existence. Only when the backend sends the first writeset, it connects to the replication manager and the state machine enters L_IDLE_STATE. From now on, whenever the replication manager receives a writeset from this backend, the state machine moves to L_SEND_STATE and the replication manager multicasts the message. If a transaction aborts after it sends the writeset, the backend has to notify the replication manager. The state machine will go back to L_IDLE_STATE. When the writeset is delivered by GCS, it might belong to an aborted transaction in which case the state is not changed. If the writeset belongs to a not aborted transaction, the state machine enters L_CHECK_STATE and the GCS channel is disabled. At this moment, the replication manager generates the GID for the local transaction, and sends it with a RECEIVED notice to the backend. Having received the feedback from that backend in form of a CHECKED message, the replication manager enables the group communication socket again. Then the state machine goes back to

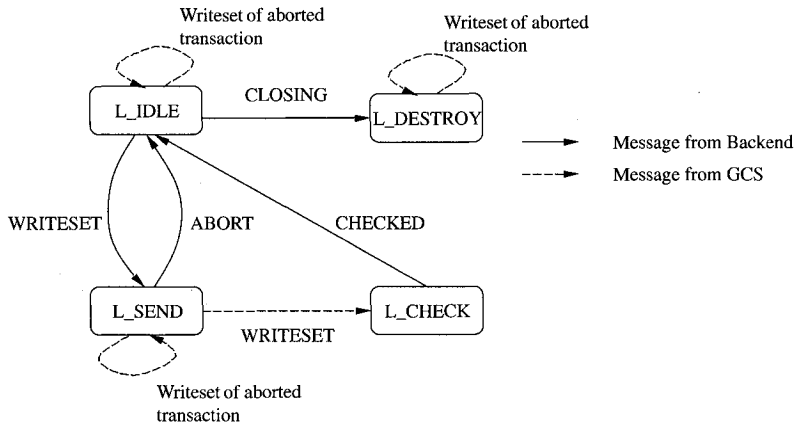

Figure 5.4: State Machine for Local Transaction

L_IDLE_STATE. When the local client logs off, a CLOSING notification will be sent to the replication manager. The state machine enters L_DESTROY_STATE. Later, this connection struct will be freed by the cleanup routine. Note that the connection struct will not be freed until all writesets for the corresponding backend have been delivered. The replication manager is not aware of the existence of a transaction in the backend until it receives the writeset for the transaction. Therefore, the replication manager does not know about read–only transactions or the transactions that abort before they send the writesets.

The remote state machine (see Figure 5.5) starts with the R_FREE_STATE indicating that this backend is ready to process writesets. When the replication manager receives a writeset from GCS, it generates the GID for this remote transaction and forwards the writeset together with the GID to the remote backend. At the same time, it blocks the socket from the GCS to guarantee serial execution. Then, the state machine enters R_BUSY_STATE. When the remote transaction fails the version check or it performs the transaction successfully, the backend notifies the replication manager. If the transaction fails, the GID will be reused for the next transaction. The state machine will go back to R_FREE_STATE.

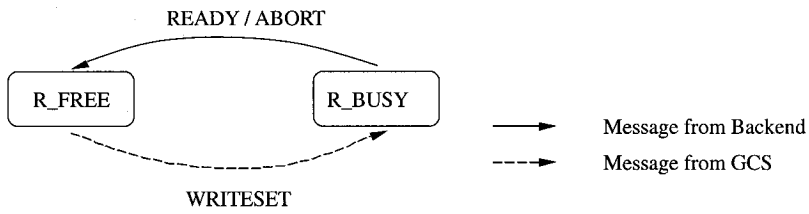

Figure 5.5: State Machine for Remote Transaction

5.2.2 Remote Transaction Aborts Local Transaction

Special care has to be taken if a remote transaction passes the version check. In this case, it might have aborted some local conflicting transactions whose writesets have been sent but not yet been delivered. The replication manager has to catch these writesets and discard them. There are three related events: the replication manager receives an ABORT message from a local backend, a READY message from a remote backend or a writeset from the GCS. We know that a READY message must be received before the replication manager receives a new writeset because the GCS channel is blocked. However, the READY message and the corresponding ABORT messages, and the ABORT messages and the corresponding writesets might arrive in any orders.

Assume a remote transaction T_1 aborts a local transaction T_2 which had already sent the writeset. We have three cases:

1. The replication manager receives T_1's READY message before T_2's ABORT message.
2. The replication manager receives T_2's ABORT message before T_1's READY message.
3. The replication manager receives T_2's ABORT message and a writeset for a new transaction T_3 from the same backend before it receives T_2's writeset.

To handle the first two cases, the replication manager must block the channel from the GCS until it has received the READY message and all of the ABORT messages for those aborted local transactions. To be able to do so, the READY message contains all TIDs of all aborted local transactions. The replication manager has a counter

keeping track of how many ABORT message it has already received and how many it needs to receive before the communication channel can be unblocked.

To solve the last problem, the replication manager must keep additional information in the `connection` struct to remember those aborted transactions for each backend. When the replication manager receives a writeset that belongs to an aborted transaction, the writeset will be discarded.

5.3 Backend

5.3.1 State Machines

Each backend also keeps track of its state using its own state machine. The state machine for a remote backend is almost the same as the state machine maintained by the replication manager (see Figure 5.5). The state changes just happen at different time points. The state switches its state from R_FREE_STATE to R_BUSY_STATE when the backend receives a writeset from the replication manager. And it switches from R_BUSY_STATE to R_FREE_STATE when the backend sends a READY or ABORT message to the replication manager.

However, the state machine of the local backend is quite different. Figure 5.6 describes the state machine when executing a transaction in the local backend. When a backend starts, it is in L_INPUT_STATE. It listens on the client socket waiting for a command. When the backend receives a command in its entirety, it switches to L_LOCAL_STATE. In this state, the backend is processing the command and collecting the writeset if there are updates. Upon receiving a commit request and the transaction running in the backend is not read–only, the backend enters L_SEND_STATE and sends the writeset to the replication manager. Otherwise, when the backend finishes executing a normal command or the transaction aborts in the middle of the process, the state machine goes back to L_INPUT_STATE. After having sent the writeset, the backend goes to L_WAIT_STATE. To maintain the integrity of the writeset message, abort is not allowed in L_SEND_STATE. When receiving a delivery notice from the replication manager, the transaction commits and the state goes back

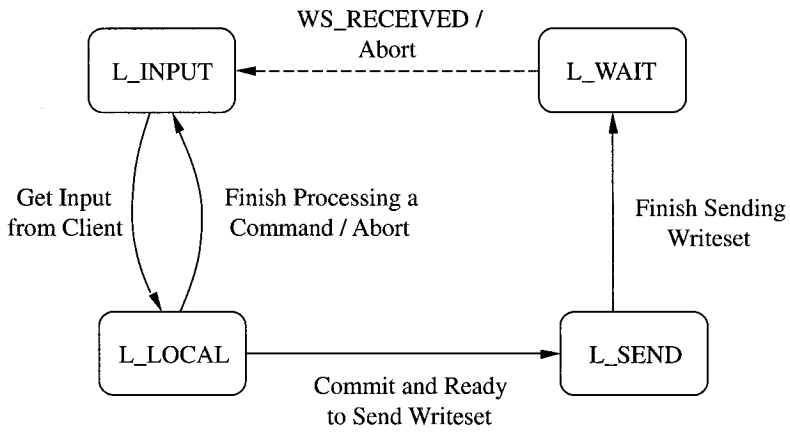

Figure 5.6: State Machine for Local Transaction

to L_INPUT_STATE. The backend might get an ABORT notice from a conflicting remote transaction. In this case, the transaction will abort and enter L_INPUT_STATE. The local backend has to notify the replication manager with an ABORT notice.

5.3.2 pg_transrecord System Table

When a transaction T (local/remote) starts, a TID is assigned locally and it is stored in the `Proc` struct of the backend as long as the transaction is active (see Section 4.1.1). TID is used throughout the execution of T. When there is a write operation, TID is logged into the WAL before T commits. When T updates a tuple, the old valid version receives TID as t_xmax and the new version takes TID as t_xmin. Also in the replicated system, TID has an important role. Furthermore, we need the GIDs. Specifically, remote transactions have to rely on GIDs to perform the version check and the locking (see Section 5.3.4). The version check requires to check for updates of transactions that have been executed on different machines with different TIDs. Hence, TIDs created by different machines can not be used for the version check, but GIDs must be used. However, PostgreSQL only stores local TIDs within the tuple. Hence, we have to match these local TIDs with the corresponding GIDs to detect conflicts.

When a backend receives the GID for the current transaction from the replication manager, it saves the GID into its `Proc` struct. However, this information is volatile since whenever a backend receives a new GID, the GID for the previous local transaction of this backend is overwritten. As a consequence, we have to store the GID, together with the associated TID, in a persistent way and accessible to all backends. There are several possible approaches. The GID can be added to each tuple in addition to the local TID. This costs a lot of disk space and requires to access tuples of local transactions twice (once when they are updated in the execution phase and once after the delivery of the writeset when the GID is determined). Alternatively, we can create a new log file, as used in the distributed recovery solution for master/slave replication (see Section 4.3). However, while in the recovery solution, the log is only needed to be visible to the replication manager, we need a data structre that is accessible to backends. A third alternative is to create a relational table that contains GIDs with the corresponding TIDs. An ever better solution is to use a system catalog to store such information. As described in Section 4.1.5, the system catalog handles data very efficiently. And this catalog can also be potentially used for distributed recovery.

Hence, a new catalog, called pg_transrecord, was created for fast retrieval. TID and GID are the attributes. Furthermore, an index on TID was created. Three functions have been implemented to add a new record for a new transaction, to update the GID for a record, and to retrieve the GID for a given TID. It is important that the pg_transrecord system table is not replicated since it contains different data in the different replicas.

When a local transaction starts, a new record is created for the transaction in the pg_transrecord. Since at this time, only the TID for the local transaction is known, the GID is set to NULL. If the writeset for the transaction is delivered, i.e. the RECEIVED message has arrived, the GID attribute is updated. For a remote transaction, the TID and the GID are added to the pg_transrecord in one step upon writeset delivery at the beginning of the transaction. Note that the visibility of the records in the catalog follows the the same rules that hold for regular tables. An active transaction can only see the records which are added/updated by transactions

that were committed before it started. Therefore, the catalog is not enough. To retrieve the GIDs for active transactions, we have to walk through the `Proc` structs in shared memory.

5.3.3 Remote Transaction Aborts Local Transaction

In our algorithm, if a remote transaction requests a lock and the lock is held by a local transaction, the local transaction must abort. It is similar to the last scenario of transaction abort in PostgreSQL, i.e. a transaction receives an asynchronous abort signal (see Section 4.1.4). It can happen at any arbitrary moment in the local transaction execution. In our implementation, the signaling mechanism is chosen to let a remote transaction abort a local transaction. To implement this functionality, the abort mechanism in PostgreSQL must be enhanced. Recall that an abort signal could not be processed immediately if ImmediateInterruptOK = false, InterruptHoldoffCount > 0 or CritSectionCount > 0 (see Section 4.1.4)). Now, we have several new situations which do not exist in the centralized system. First, when the writeset is sent, the backend will wait for the writeset delivery confirmation. In this case, we want the backend to stop waiting if there is an abort signal. Second, when a local transaction is ready to commit, the writeset for the transaction will be sent to the replication manager. Here, we do not want the transaction to be interrupted in the middle of the transmission. Third, a transaction can not be aborted when it is waiting for the input from a client or in the middle of input transmission. Otherwise, partial client requests might be left in the communication channel. Note that 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.

To handle the first case, a local transaction will abort immediately when it is in L_WAIT_STATE. In order to handle the last two cases, two additional fields have been added to the `Proc` struct of a local transaction. A boolean field *AbortFlag* is set to true if a local transaction catches an abort signal from a remote transaction but it can not abort immediately, because it is not safe. These unsafe situations include the

unsafe situations in the centralized system (Section 4.1.4) as well as the last two new cases we described. Another field *NoAbortDelay* is added to guide against the last new situation. In summary, in our replicated system, a local transaction is aborted (through signal by a remote transaction), if state = L_WAIT_STATE or (ImmediateInterruptOK = true and InterruptHoldoffCount = 0 and CritSectionCount = 0 and AbortFlag = true and NoAbortDelay = true).

It is possible that a remote transaction sends an abort signal to an aborting transaction. In this case, PostgreSQL only aborts a transaction once (see Section 4.1.4). Recall that, there is only one remote transaction or one local transaction processing a writeset, since the replication manager will not deliver the next writeset until it receives the confirmation that the previous transaction has terminated (and local transactions aborted by a remote transaction have aborted). Hence, the complicated situation, in which two remote transactions send abort signals to an aborting local transaction which has already sent the writeset, is avoided.

5.3.4 Version Check and Execution of Remote Transactions

Preparing for Version Check

Version check for a local transaction is the same as it is in the original PostgreSQL. In the execution phase, the local TID is used to do the version check. When the version check is successful, the operation will be performed. Otherwise the transaction aborts. The extra work is that if there is a write operation of transaction T_i and the operation is successfully performed, we have to collect the GID corresponding to the *t_xmin* of the version V which is read and copied by T_i and then invalidated by setting *t_xmax* to TID_i. This GID can be retrieved from the *pg_transrecord* given *t_xmin*. That is, for a local transaction T_i which successfully performs a write operation on a tuple X. If t_xmin of the valid version of X is TID_j, we get GID_j corresponding to TID_j by looking up the pg_transrecord.

Then we have to attach to the writeset for each tuple the corresponding GID. Recall that there is a `tuple collection` data structure within the writeset which contains all the modified tuples by a query. An array, *GIDArray* is added to the

`tuple collection`. Each entry within GIDArray matches the corresponding tuple in the `tuple collection`. If the changes of a tuple X are kept in the k^{th} position in the `tuple collection`, GID_j is also kept in the k^{th} position in the GIDArray.

Version Check

Now assume a writeset is delivered and a remote transaction T_i with TID_i, corresponding to the local transaction T_i' on the local site, is started. Assume T_i' accessed tuple X and read version created by T_j with TID_j and GID_j. GID_j was added to the writeset. At the remote site, T_i now performs the version check. It reads the valid version of X with $t\_xmin = TID_k$ with corresponding GID_k. If $GID_k = GID_j$, T_i passes the version check for the tuple.

Execution

In the version check and early execution phase, when a remote transaction finds there is no one holding the data object, it will get the lock and process the operation right away instead of waiting until the late execution phase. However, if a local transaction holds a lock, we do not immediately abort it and execute, but wait until the check is complete. When the remote transaction has passed all version checks, it will go back and perform the rest of the updates which have not been done in the early execution. For that, it will again go through the `tuple collection` one by one. Now it has to detect which tuples were already updated and which not. To determine this, the GIDArray is re–used. After an update operation has been processed in the early execution phase, the entry in the GIDArray is set to a special value $DONE$ (a negative number which is not a legal GID). So in the late execution phase, if the GIDArray entry equals DONE, the operation on the tuple has already been performed and will be skipped this time.

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 the late execution phase. As we

have described (see Section 4.1.3), in PostgreSQL the lock holder wakes up all of the transactions waiting in the queue to let them compete for the lock. The order in the original waiting queue is totally irrelevant to who might finally be the new holder. This is not what we want. Our strategy is that we put the remote transaction at the head of the waiting queue and do not wake up any local transaction until the remote transaction holds the lock. First, when there is a lock request from a remote transaction and the holder is a local transaction (by checking whether it has already a GID), this request will be put at the head of the waiting queue without any further conflict check within the waiting queue. Then the process in which the remote transaction is running, sends an abort signal to the holder process and goes into sleep. As the holder catches and processes the signal, it only wakes up the remote transaction, which is the first process in the waiting queue. The rest of the waiting queue is passed to that remote transaction. When the remote transaction wakes up, it holds the lock and then wakes up the rest of the processes in the waiting queue to continue with the standard PostgreSQL procedure.

Another note is that the only reason causing a remote transaction to abort is failing a version check. So the remote transaction should never invoke the deadlock detection routine. It is achieved by skipping the timer setting for the deadlock detection.

Chapter 6

Evaluation and Discussion

We evaluated the performance using two different applications. The first test suite uses a TPC-W benchmark variant to simulate a real-world application. The second test suite uses a 100% update workload. 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.

6.1 TPC-W Benchmark

We expect our system to have good performance in real-world applications since they are mainly read intensive, and snapshot isolation favors read-only transactions. We performed our tests using the OSDL-DBT-1 benchmark [23]. It is a simplified version of the TPC-W benchmark [14] simulating an online bookstore. There are three different workload types by varying the ratio of browsing to buying transactions: primarily shopping, browsing and ordering. In our experiment, we choose the browsing workload, which contains 80% browsing transactions 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

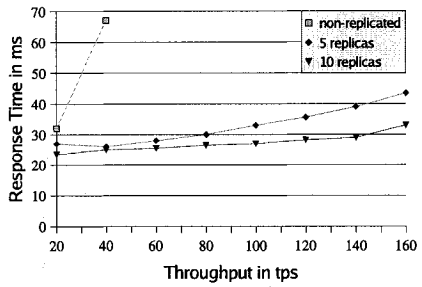

Figure 6.1: TPC-W: Browsing (read-only)

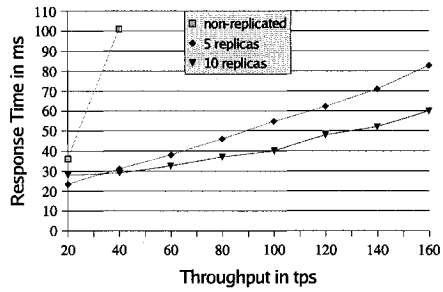

Figure 6.2: TPC-W: Ordering (update)

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. The throughput in transactions per second (tps) is controlled by the think-time parameter, i.e., the time a client waits between two consecutive requests.

We run the experiment with a centralized, non-replicated server, and then with 5 and 10 replicas. Figure 6.1 shows the client response time for browsing transactions, and Figure 6.2 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

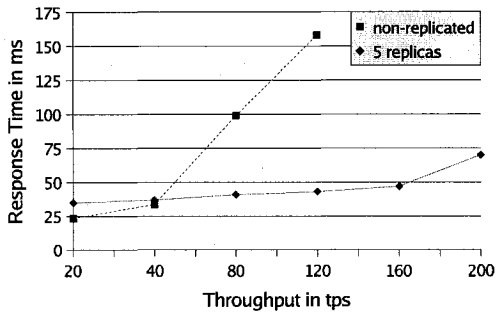

Figure 6.3: Update Workload: Response Time

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 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 [21].

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.

6.1.1 Update Intensive Workloads

The second experiment only uses update transactions. In this case we would not expect any performance gain compared to a centralized case since all updates are executed at all replicas. In contrast, we would expect higher response times because of the total order multicast, write set collection, the overhead of the RM, etc. The database consists of 10 tables with each 1000 tuples. Each update transaction consists of 10 operations each updating exactly one tuple (randomly chosen from the 10000 tuples). There are 20 clients in the system each submitting transactions with a rate as to achieve a certain system throughput. Figure 6.3 shows the response time with increasing load.

At low throughputs, the central system has faster response time due to the replication overhead for update transactions. However, to our surprise, 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 (think-time) it looks like that they put a considerable administrative burden on the system. We tried to put the clients on another machine in the LAN with the same results. Another minor reason might be that in the replicated case, only one replica executes a transaction, the others only apply the changes which takes less time. This leaves more resources free to execute additional transactions. However, the difference is not big enough to explain the results of the figure. As a summary, the advantages of distributing clients over several replicas provides performance gains that are higher than any possible disadvantage of replication. We are currently investigating whether a smarter client management might improve the situation for a central server, and whether we can build a test suite where the disadvantage of the serial execution of remote transactions becomes more apparent.

6.1.2 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, not only in their concurrency control component, but in many other modules. For instance, the buffer management (FORCE vs. NOFORCE), and client management is different. In general, however, the relative performance of both approaches is similar. This proves that the general replica control approach (executing transactions locally, sending writesets at the end of the transaction using a GCS, and applying writesets efficiently at the remote replicas) is a good way to provide high throughput and scalability in a LAN setting.

Although other middleware based approaches evaluate their systems using the TPC-W benchmark [4, 25], we think a direct comparison is unfair since the setups are always quite different (implementation of the benchmark code, client setup, database size, etc.).

Chapter 7

Discussion of Optimization and Conclusion

7.1 Discussion of Optimization and Future Work

We have discussed the current version of our replicated database system. Now we discuss some possible optimizations. In our solution, the processing of remote writesets is serial. We have some room to improve our algorithm in this regard. As we have described, the replication manager does not deliver the next writeset until the transaction, which owns the last delivered writeset, commits or aborts. One possible improvement is that the replication manager could deliver the next writeset as soon as the version check for the previous writeset is completed. To do that, we have some problems to solve:

- In our protocol, if a lock is not granted in the version check and each eaecution phase, the remote backend will not wait for the lock. To allow concurrent execution of remote backends, there should be a locking mechanism to order conflicting operations of remote transactions.
- In this case, a remote transaction must be able to wait on several locks at the same time. The locking protocol has to be extended accordingly.
- The replication manager has to coordinate the remote transactions and the local aborted transactions, whose writesets have been sent. Now, we can have multiple

remote transactions in progress. This coordination protocol has to be analyzed.

- Another potential issue is that GID's can not be guaranteed to be continuous anymore. There is no problem for concurrency control. However, it might add extra work to the distributed recovery protocol.

There are many occasions that one of the nodes can fail. For example, a node might crash. Or one of the servers is slow and causes the buffer in the GCS overflow. In our system, such a node is automatically excluded by the GCS and the virtual synchrony property of the GCS guarantees that the other nodes can continue as if nothing has happened. However, failed nodes should be restarted and again added to the system. Hence, a recovery module has to be added. The recovery model of the master/slave version of Postgres–R is not working any more with the new version since the flow control in the replication manager changed considerably. Also, there is no master anymore. However, we believe that with little modifications, we can have recovery again in our system. We also can improve the original distributed recovery mechanism, since we can use the pg_transrecord system catalog in the database rather than the distributed recovery log of the replication manager.

7.2 Conclusion

This thesis presents the design and implementation of a synchronous and update everywhere database replication approach based on Snapshot Isolation. Our experiments show that this approach has good performance. This work also demonstrates that synchronous and update everywhere are feasible, at least in a cluster of workstations within a LAN environment. As the algorithm matches the original concurrency mechanism in the PostgreSQL database management system, this work can be an excellent extension of PostgreSQL. The project is published as an open–source development project to integrate replication solutions into PostgreSQL. It is available at http://gborg.postgresql.org/project/pgreplication/.

Bibliography

[1] M. E. Adiba and B. G. Lindsay. Database Snapshots. In *VLDB*, pages 86–91, 1980.

[2] Y. Amir and J. Stanton. The Spread Wide Area Group Communication System. Technical Report CNDS 98-4, Center of Networking and Distributed Systems, Johns Hopkins University, 1998.

[3] Y. Amir and C. Tutu. From Total Order to Database Replication. In *Int. Conf. on Distr. Comp. Systems (ICDCS)*, 2002.

[4] C. Amza, A. L. Cox, and W. Zwaenepoel. Distributed versioning: Consistent replication for scaling back-end databases of dynamic content web sites. In *Middleware*, 2003.

[5] W. Bausch. Integrating Synchronous Update–everywhere Replication into the PostgreSQL Database. Master's thesis, Swiss Federal Institute of Technology in Zürich, March 1999.

[6] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A Critique of ANSI SQL Isolation Levels. In *ACM SIGMOD Int. Conf. on Management of Data*, pages 1–10, June 1995.

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

[8] K. Birman, A. E. Abbadi, W. C. Dietrich, T. Joseph, and T. Raeuchle. An Overview of the ISIS Project. January 1985.

[9] K. P. Birman and T. Joseph. Exploiting Virtual Synchrony in Distributed Systems. In *the 11th ACM Symposium on Operating Systems Principles*, pages 123–138, November 1987.

[10] M. Chouk. Master–Slave Replication, Failover and Distributed Recovery in PostgreSQL Database. Master's thesis, McGill University, June 2003.

[11] Borland Software Corporation. Interbase Documentation, 2004.

[12] Oracle Corporation. Oracle's Solutions for the Distributed Environment, June 2002.

[13] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Design.* Addison Wesley, 2000.

[14] Transaction Processing Performance Council. TPC Benchmark W, 2000.

[15] D. Dolev and D. Malki. The Transis Approach to High Availability Cluster Communication. *Communications of the ACM*, 39(4):63–70, April 1996.

[16] R. Goldring. A Discussion of Relational Database Replication Technology. *InfoDB*, 8(1), 1994.

[17] J. Gray, P. Helland, P. E. O'Neil, and D. Shasha. The Dangers of Replication and a Solution. In *ACM SIGMOD Int. Conf. on Management of Data*, pages 173–182, 1996.

[18] The PostgreSQL Global Development Group. PostgreSQL 7.2 Documentation, 2001.

[19] Theo Härder. Observations on Optimistic Concurrency Control Schemes. *Inf. Syst.*, 9(2):111–120, 1984.

[20] J. Holliday, D. Agrawal, and A. El Abbadi. The Performance of Database Replication with Group Communication. In *IEEE International Symposium on Fault Tolerant Computing (FTCS29)*, pages 158–165, 1999.

[21] 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.

[22] B. Kemme and G. Alonso. A New Approach to Developing and Implementing Eager Database Replication Protocols. *ACM Transactions on Database Systems (TODS)*, 25(3):333–379, September 2000.

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

[24] L. E. Moser, Y. Amir, P. M. Melliar-Smith, and D. A. Agarwal. Extended Virtual Synchrony. *The 14th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 56–65, June 1994.

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

[26] M. Stonebraker and G. Kemnitz. The postgres next generation database management system. *Commun. ACM*, 34(10):78–92, 1991.

[27] R. van Renesse, K. P. Birman, and S. Maffeis. Horus: A Flexible Group Communications System. *Communications of the ACM*, 39(4):76–83, April 1996.

[28] J. Worsley and J. Drake. *Practical PostgreSQL*. O'Reilly Media, Inc., 2002.