

Online Recovery in Cluster Databases

WeiBin Liang
wliang2@cs.mcgill.ca

Bettina Kemme
kemme@cs.mcgill.ca

School of Computer Science, McGill University
Montreal, Canada

ABSTRACT

Cluster based replication solutions are an attractive mechanism to provide both high-availability and scalability for the database backend within the multi-tier information systems of service-oriented businesses. An important issue that has not yet received sufficient attention is how database replicas that have failed can be reintegrated into the system or how completely new replicas can be added in order to increase the capacity of the system. Ideally, recovery takes place online, i.e., while transaction processing continues at the replicas that are already running. In this paper we present a complete online recovery solution for database clusters. One important issue is to find an efficient way to transfer the data the joining replica needs. In this paper, we present two data transfer strategies. The first transfers the latest copy of each data item, the second transfers the updates a rejoining replica has missed during its downtime. A second challenge is to coordinate this transfer with ongoing transaction processing such that the joining node does not miss any updates. We present a coordination protocol that can be used with Postgres-R, a replication tool which uses a group communication system for replica control. We have implemented and compared our transfer solutions against a set of parameters, and present heuristics which allow an automatic selection of the optimal strategy for a given configuration.

1. INTRODUCTION

Over the years, we have witnessed an increasing demand for scalable and highly reliable information systems. In particular in the emerging service economy, businesses have to provide continuous access to their services to both customers and trading partners. With an increasing number of clients the IT infrastructure of these businesses is facing immense scalability and availability requirements. This infrastructure typically consists of a multi-tier architecture with a web-server tier providing the presentation logic, an application server tier providing the business logic, and a database tier providing persistence for business critical data.

For many applications, the I/O intensive database backend is the performance bottleneck of the system. An attractive way to provide scalability and at the same time increase the availability of this tier, is to have a cluster of machines and replicate the database on these machines, i.e., to have several database instances each having a copy of the database. When the workload increases, we can scale up the system by simply adding more replicas to the cluster. Furthermore, if one replica fails, its workload can be taken over by another replica, providing fault-tolerance. Most replication approaches follow a read-one/write-all approach reading data from one replica but performing write operations on all replicas. In recent years, many cluster based replication solutions have been proposed showing excellent performance in terms of low response times and high throughput [1, 2, 4, 6, 7, 10, 14, 17, 20, 21, 22, 23, 24, 28].

Reconfiguration in cluster databases refers to the fact that replicas can both leave and join the cluster. In most of the existing approaches, when a replica fails, the other replicas are informed appropriately and currently executing transactions will not need to perform their writes on the crashed replica. That is, they follow a read-one/write-all-available approach. The protocols further guarantee that the transactions committed at the crashed replica are a subset of transactions committed at the available replicas. Some approaches even transparently redirect clients that were connected to the crashed replica to an available replica.

Adding replicas efficiently and correctly to the system is as important as proper failure handling. In order to guarantee availability, failed sites have to be rejoined to the system. Furthermore, it must be possible to add new sites to the system in order to handle higher demands. In the following, we refer to reconfigurations where replicas *join* the system as *recovery*, although completely new replicas do not recover in the strict sense, since they had not failed. Only few replication solutions had a closer look at recovery and we are not aware of any proper overhead analysis. We discuss them in the related work section.

The recovery process can be divided into two steps. First, upon restart of a failed replica, *local recovery* brings the database of the failed site back into a consistent state. This is standard database technology which is also needed for non-replicated database systems. It requires to potentially redo some of the transactions that committed at the replica before the crash and undo transactions that were aborted or active at the time of the crash. A new replica does not need to perform this step. To perform local recovery, database systems usually maintain a log (also referred to as write-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

EDBT'08, March 25–30, 2008, Nantes, France.

Copyright 2008 ACM 978-1-59593-926-5/08/0003 ...\$5.00.

ahead-log) during normal processing such that for each write operation on a data item x , the before- and after-image of x is appended to the log. Further, it contains information about which transactions committed and aborted. Local recovery is not the focus of this paper. Details can be found, e.g., in [13].

After local recovery, *distributed recovery* provides the joining node with the current state of the database. It has to reflect the updates of all transactions that committed during the downtime of the joining node. Typically one of the available nodes in the system is responsible for this *data transfer*. We explore two different data transfer mechanisms. Firstly, a peer site can transfer the updates the joining site has missed during its downtime to the joining site where they are applied. The second approach takes a snapshot of the data of a peer site and installs it at the joining site. For completely new replicas that are added to the system, only the second approach can be used. In this paper we explore the significant factors that affect the performance of each strategy, and how to choose one strategy over the other. In particular, this paper presents a hybrid recovery mechanism that is able to perform both types of recovery. Based on heuristics, it dynamically chooses the strategy for which a faster recovery is expected.

Distributed recovery can be performed both offline and online. Offline recovery means that the system stops processing client requests while a site is being brought into the system. The system only resumes its normal operations once recovery is done. Thus, the system is non-responsive during recovery which is not acceptable for high-available, high-throughput systems. Online recovery, in contrast, allows the system to continue processing client requests even during the recovery of sites. However, it requires recovery to be synchronized with the execution of ongoing transactions at other sites. A *recovery coordination protocol* has to guarantee that the joining site does not miss the updates of any transaction that is executing while the data transfer is ongoing. For that, however, the recovery algorithm has to be aware of how transaction execution is controlled in the replicated system, i.e, it has to be adjusted to work with the replica control mechanism in place. In this paper, we focus on replication in Postgres-R [28], which provides replication for the open-source PostgreSQL database system. In Postgres-R, a transaction is first executed at the local replica to which it is submitted. At commit time, the local replica multicasts the updates of the transaction in form of a write-set to the other replicas. By using a total order multicast provided by group communication systems (GCS) [9], each replica receives writesets from different transactions in exactly the same order. This total order is used to serialize the transactions across the entire cluster. Our recovery coordination protocol also takes advantage of the properties of the GCS in order to synchronize recovery with ongoing transaction processing and to determine when a joining site is up-to-date.

The contributions of this paper can be summarized as follows.

- We present two data transfer strategies for a replicated database system that provide a crashed replica or a completely new replica with the current state of the database. The strategies can be used with all replication solutions that are based on the read-one/write-all-available approach.

- We provide a recovery coordination protocol that facilitates that transactions can continue to execute in the system during the recovery of a replica. The recovery coordination protocol is appropriate for replication approaches that use a group communication system for replica communication. Our recovery coordination protocol is combined with both data transfer strategies and implemented in Postgres-R.
- We have performed an extensive performance evaluation of our implementation and provide guidelines that show when each of the approaches works best.

The rest of the paper is structured as follows. Section 2 gives an overview of Postgres-R and its concurrency and replica control algorithms. Section 3 provides a short introduction to group communication systems. Section 4 presents our recovery solution. It both presents the solution in algorithmic form and discusses some interesting implementation details. Section 5 gives an evaluation of this implementation and compares the two recovery strategies. Finally, Section 6 presents related work and Section 7 concludes the work.

2. POSTGRES-R

In this section we give an overview of the Postgres-R system. Although only the recovery coordination protocol and not the data transfer strategies are dependent on the underlying replication tool, we present Postgres-R first in order to give the reader a good understanding of the principles of transaction execution in a replicated database system.

Postgres-R [28] is an extension of the open-source database system PostgreSQL. PostgreSQL uses multi-version concurrency control providing snapshot isolation (SI) [3] as its highest isolation level. SI has become popular in recent years, being also provided by Oracle and SQL Server. Each update of transaction T_i on data item x creates a new version x_i of x . A transaction T_i reads the version of x created by transaction T_j such that T_j committed before T_i started, and there is no other transaction T_k that also wrote x , commits after T_j but before T_i starts. That is, T_i reads from a committed snapshot of the data as of T_i 's start time. However, if T_i also updates x , then it later sees its own updates. Conflicts are only detected between write operations of concurrent transactions. Two transactions T_i and T_j are concurrent if neither tcommitted/aborted before the other started. SI disallows that two concurrent transactions update the same data item x and both commit.

In Postgres-R each replica maintains a full copy of the database. Each replica accepts both read-only and update transactions. A transaction T_i is first executed at the local PostgreSQL replica R it is submitted to. We say T_i is local at R and R is T_i 's local replica. R uses its local concurrency control mechanism to isolate T_i from other transactions running concurrently at R . When the application submits the commit request the replication module of R sends a writeset containing all records T_i has changed and multicasts it to all replicas in the system. For that purpose it uses a group communication system (GCS) [9], which provides a total order delivery. That is, while different replicas might multicast writesets concurrently, the GCS will deliver to all replicas all writesets in exactly the same order. It will also deliver a writeset back to the sender itself. We will talk about the exact properties of GCS shortly. When a replica receives a writeset for a transaction T_i it performs validation. It

checks whether there was any transaction T_j such that T_j 's writeset was delivered before T_i 's writeset, T_i is concurrent to T_j , T_j passed validation, and T_i and T_j conflict (i.e., update the same data item). If yes, then T_i does not pass validation and may not commit. At T_i 's local replica R , T_i is aborted, at the remote replicas the writeset is simply discarded and T_i never started. If T_i passes validation, then R simply commits T_i . At the other replicas, T_i is started as a remote transaction, and the updates listed in the writeset applied. The replication scheme guarantees that each replica validates transactions exactly in the same order (the total order determined by the GCS), and makes the same commit/abort decision for all transactions. Furthermore, transactions are committed at all replicas in the order in which their writesets are delivered, hence, the commit order is the same at all replicas.

The validation mechanism in Postgres-R has to guarantee that snapshot isolation is guaranteed across the entire replicated system. During local execution of a transaction T_i , the local replica provides T_i with a snapshot and detects conflicts between other transactions that are currently executed at this site. The validation phase after writeset delivery detects conflicts between transactions with different local replicas. For this, a proper transaction identification is needed. A transaction T_i actually creates one independent transaction at each replica: a local transaction at the replica T_i is submitted to (which executes all SQL statements), and remote transactions at the other replicas (which apply the writeset). Each replica creates its own internal PostgreSQL transaction identifier TID. This makes it hard to compare two logical transactions T_i and T_j and determine whether they are concurrent. Therefore, a global transaction identifier GID is needed, that links the transactions that are executed at the different replicas on behalf of a logical transaction T_i . For this purpose, each replica keeps a GID counter. Whenever a writeset is delivered, the counter is increased and the current value assigned as GID to the corresponding transaction. Since each site delivers writesets in the same order, all replicas assign the same GID value to a given transaction. Each replica keeps an internal table that allows for a fast matching between the internal identifier TID for a transaction and its corresponding GID.

The details of how validation is performed, how the write sets of remote transactions are applied, and how they are isolated from concurrently running local transactions are not important in the context of this paper, and we refer the interested reader to [28]. In fact, the ideas behind the coordination protocol presented in this paper are likely to work with many replica control solutions based on GCS, probably with only small adjustments.

3. GROUP COMMUNICATION SYSTEMS (GCS)

In this section, we shortly introduce the properties of a GCS that are important in the context of this paper. GCS [9] manage groups of processes. A group member can multicast a message to the group which is then received by all members including the sender. GCS provide different ordering guarantees. In this paper, we are only interested in the total order multicast where the GCS delivers the same stream of messages to all group members. GCS provide different levels of reliability in regard to message delivery. In

here, we assume *uniform-reliable* delivery. It guarantees that whenever a member receives a message (even if it fails immediately afterwards) all members receive the message unless they fail themselves. With this, it is impossible that a replica sends a writeset, receives it back, commits the corresponding transaction and then fails, while none of the other replicas has received the writesets. Instead, if one replica receives a writeset, all other available replicas will also receive the writeset. Thus, the transactions committed at a failed replica are always a subset of the transactions committed at the available replicas.

GCS also provide advanced group maintenance features. A GCS automatically removes crashed members from the view of currently connected members. The available members receive a view change message V containing the list of current members of the group. Similar view change messages are delivered if a site explicitly joins or leaves the group. Group reconfiguration provides the *virtual synchrony* property: if members p and q receive both first view V and then V' , then they receive the same set of messages while members of V . This means, if p receives a view change message V , then a set of messages, and then a new view change message V' containing a new member q , then p knows that q had not received any message delivered after V but before V' but it will receive the messages delivered after V' (unless a new view change occurs). In an asynchronous environment (no bounds on message delay), the GCS might wrongly exclude a non-crashed member. In this case, Postgres-R requires the affected replica to shut down and initiate recovery.

4. DISTRIBUTED RECOVERY

4.1 Data Transfer Strategies

We are looking at two different data transfer strategies: the Total Copy Strategy (TCS) and the Partial Copy Strategy (PCS). With TCS, a peer site S_p sends a complete copy of its data to the joining site S_j . With PCS, only the updates of committed transactions (i.e., the writesets) that the joining site missed during its downtime are transferred. Hence, in order to use PCS, the running system must maintain logs that keep this writeset information. This could lead to additional overhead during normal processing. However, it might be possible to use the standard redo log – used for local recovery – to extract the writeset information such that there is no additional logging due to distributed recovery. When a new site joins a running system, PCS cannot be applied unless the current system keeps the writesets of all transactions that have been committed since the original start of the system, which is very unlikely. When a crashed site rejoins the system, both TCS and PCS can be used.

In general, one can assume that when the database is small or if the joining site S_j has been down for a long time and has missed many transactions, then TCS will outperform PCS. This is true because in this case, TCS sends less data than PCS. Furthermore, if the log of the peer site S_p does not contain all the writesets that are needed by S_j , TCS must be used. This can happen, if S_p truncated its log, or if S_p joined itself via TCS and only after S_j had failed. Also, one must keep in mind that PCS requires to log the update information which might result in increased overhead during normal processing.

However, if the database size is large or the joining node was down for only short time and has only missed few trans-

actions, then PCS will transfer less data than TCS, and hence outperform TCS. Hence, our system aims in a flexible solution that allows at runtime to choose the appropriate transfer strategy according to the configuration.

There exist alternatives to these data transfer strategies, and we want to discuss them shortly. In case that a failed site rejoins the system, data transfer would be minimized if only the data items that were changed during the downtime are transferred to and updated at the recovering site. This is very attractive when many transactions were executed during the downtime of the recovering site, but they only changed a small active part on a huge database (i.e., hot-spot data). In this case, PCS has to apply many transactions, although for each data item only the last change is relevant. TCS will copy a lot of data that has actually not changed. We will discuss later how such an approach can be implemented as an extension of PCS. Another alternative would be to transfer the latest version of a data item to the joining node only when it is needed for the first time or when the next update on the data item occurs. However, keeping track of what has already been updated can be quite complicated in a relational database system where the granularity of a data item is a record. Also, requesting individual records whenever needed during query execution is a complex process and might result in high delays for queries if they have to wait for a set of records to arrive. Thus, we do not further consider this alternative.

4.2 Synchronization

If recovery is performed completely offline no node may execute transactions while the joining node receives the data. This is clearly undesirable. Using online recovery, the rest of the system continues executing transactions. The question is how to handle these transactions. There are basically two options: the updates of these transactions are included in the state transferred by the peer to the joining site or the joining site applies them after recovery is completed as standard remote transactions. In any case, some synchronization protocol has to decide for a given transaction T which of the two options is used. In our case, we take advantage of the total order of update transactions determined by the GCS. Peer and joining site decide on a delimiter transaction T such that updates of missed transactions delivered and committed before T are reflected in the data transferred from the peer site to the joining site as part of recovery while T and all transactions that committed after T are applied at the joining site as standard remote transactions.

4.3 Algorithm Overview

Our distributed recovery algorithm for Postgres-R combines both TCS and PCS. The algorithm uses a heuristic to choose the best strategy to perform the distributed recovery. We refer to the writeset of a transaction as WS, and to the log containing the WSs of committed transactions as the *Writeset Log* or *WSL*. Each entry consists of a writeset WS and the *GID* of the corresponding transaction. We allow a replica to delete the oldest update information so that *WSL* does not grow indefinitely.

The distributed recovery begins when a new site joins a running system or after a previously failed site restarted and finished local recovery. This joining replica S_j first joins the GCS group composed of sites of the running system. Then, S_j locates one of the group members S_p as the peer that will

assist the distributed recovery process. Most communication between S_j and S_p is via a direct connection and not via GCS. S_j can locate potential peer sites by either looking at the membership of the current GCS or by being provided by a list of candidates via a configuration file specified, e.g., by the system administrator. S_j asks each potential candidate whether it is willing to become the peer. A site can deny the request because it is heavily loaded, for instance. After the connection is set up to a willing peer S_p , S_j informs S_p about the *GID* (*lastGid*) of the last transaction that it committed before it failed. If S_j is a new site, it sends -1 as predefined value. Recall that transactions are committed at all sites in exactly the same order and with monotonically increasing *GID*. When S_p receives *lastGid* it looks in its WSL for an entry tagged with *lastGid*. If no entry can be found, TCS has to be used. This can happen because *lastGid* is set to -1, because S_p truncated its log, or because S_p itself joined the system after S_j failed. Hence, S_p does not have the update information of all committed transactions that S_j missed, and thus, TCS is needed. If an entry is found, both TCS and PCS can be used. S_j estimates the costs for TCS and PCS recovery and chooses the one with smaller estimated cost. S_p retrieves the maximum *GID* *maxGid* contained in its WSL. The difference between *lastGid* and *maxGid* represents the number of committed transactions missed by S_j so far. If the difference is greater than a threshold, which is determined by a heuristic, it is assumed TCS will outperform PCS, and TCS is chosen. If the difference is smaller than the threshold, PCS will be used.

4.3.1 PCS

If PCS is used, S_p retrieves all WSs whose *GID* are greater than *lastGid* from WSL and sends them to S_j in multiple rounds. At each round, S_p packs a certain number of WSs into one message and sends it to S_j . Upon receipt of the message, S_j unpacks the WSs one by one from the message and applies them. After all WSs in the message have been processed, S_j notifies S_p to send more WSs.

Applying a WS for recovery is basically the same as applying a WS of a remote transaction. In principle, no locks need to be held since there are no concurrent transactions, and no version checks are needed, since there are no concurrent transactions. However, we do not distinguish between applying WSs during recovery and during normal processing for simplicity of description.

If recovery were offline, i.e., when no transaction processing were done in the system during recovery, recovery would simply be done when all missed writsets are sent to S_j and applied. In online recovery, new transactions are simultaneously executed in the running system. Since both S_j and S_p are members of the group, they receive the corresponding WSs. S_p applies the WSs and adds them to its WSL, and hence, potentially also sends them to S_j during the data transfer. Thus, S_j may receive the same WS both from S_p and the GCS. It needs to know when and how it should switch from applying WSs received from S_p to verifying and applying WSs received from the GCS. S_p needs to know up to which WS it should send to S_j . This means, a synchronization protocol is needed here. In our protocol, when S_p perceives that there are only a few more WSs left to be sent to S_j , it multicasts a special message in total order to all sites. That is, this message is ordered along with all the writesets multicast in the system. S_p is now responsible for

transferring the WSs received before this special message in total; and S_j starts to buffer WSs received from GCS after the receipt of the special message. The switch will then be straightforward. When S_j has applied all WSs from S_p , it starts to apply the buffered WSs as standard remote transactions. When all buffered WSs have been applied, the recovery is done, and S_j can start to act as a normal site and to handle client requests.

4.3.2 TCS

Our TCS solution takes advantage of the snapshot based concurrency control provided by PostgreSQL. Our TCS data transfer starts a read-only recovery transaction TS on the peer site S_i that takes a snapshot of the current state of its local database and transfers it to S_j . According to snapshot isolation this state contains the updates of all transactions that committed before TS started. Transactions can continue to execute and commit at S_p while TS reads the data. It is guaranteed that their updates are not reflected in the snapshot transferred to S_j . We keep track of these transactions, and transfer their writesets after S_j has installed the database copy. This basically means that a TCS is followed by a short PCS that transfers these missing writesets up to the delimiter transaction.

When S_p starts TS , it keeps track of $maxGID$, the GID of the last transaction that committed before TS started. It sends it to S_j which then leaves the group. S_j stores the result of TS into a file. After TS has completed, the system administrator copies this snapshot data file from S_p to S_j , restarts S_j in non-replication mode, and installs the snapshot into its local database. Note that it is important to use the non-replicated mode as otherwise installing the snapshot would trigger the system to propagate the changes to the other replicas. After the snapshot is installed, S_j contains data that is consistent with the data of S_p when the snapshot was taken. The system administrator restarts S_j again in replication mode. S_j joins the system, performs distributed recovery using PCS, with the $maxGid$ received previously from S_p as its $lastGid$. This will transfer all updates that were made while and after TS was executed. A script could replace the tasks of the system administrator.

For both TCS and PCS, clients may not connect to S_j during recovery, since it does not yet have the up-to-date state of the database. S_p , however, may execute local and remote transactions as any other non-recovering site during the recovery process.

4.4 Detailed Description

We first present the recovery algorithm that uses only PCS, and then show the extension needed to allow for both PCS and TCS. Descriptions of some parameters and functions used in the algorithm can be found in Table 1.

4.4.1 PCS algorithm

Figure 1 shows the recovery steps at the joining (recovering) site S_j , and Figure 2 shows the recovery steps at peer site S_p for PCS. As shown in Figure 1, the joining site first performs local recovery, sets up some variables and then joins the replication group. In our system, the list of potential peers in line 8 is read from a configuration file which allows the administrator to set the preferred peer site to be the first to be contacted. Alternatively, S_j could simply use the view change message delivered by the GCS to

Parameters/Functions	Description
N	The number of WSs in one message.
$Nthreshold$	determines whether to use PCS or TCS
$send(receiver, type, content)$	Sends a message.
$recv(sender, type, content)$	Receives a message.
$mcast(type, content)$	Multicasts a message.
$mcast_recv(type, content)$	Receives a multicast message.
$applyWS(WS)$	Performs the updates captured in the WS. This includes logging the WS itself
$endAssistance()$	cleanup of data structures at peer site

Table 1: Paramters, variables and functions

determine the potential peers. Once the joining site finds a peer site that is willing to transfer the data, it sends it the GID of the last transaction committed before the crash. If no site is willing to serve as peer, recovery fails (lines 8-18). During recovery, clients may not connect to S_j (lines 20-21). WSs that are received from the GCS are ignored (lines 23-24), or, if recovery is close to be finished, they are buffered (lines 25-26). S_p receives writesets from S_j in the set NWS . After applying them, it requests more from S_p (lines 27-29). When S_j receives the MSG_SYNC message, it knows that recovery is nearly done and it has to start buffering WS coming from the GCS (line 30-32). After S_j receives the $MSG_RECOVERY_DONE$ message from S_p and processes all WSs that are included in the message (line 33-34), a transitional phase starts (line 35-38). S_j first blocks the communication channel with GCS, that is, does not listen to the messages from GCS. It then processes all WSs that were buffered during the recovery phase, and then reopens the communication channel again. So, during this transitional phase, WSs that are multicast by other sites are buffered in the GCS. This simplifies synchronization.

We could have chosen a simpler synchronization point by letting S_j start buffering messages immediately after it receives the view change message indicating that it has successfully joined the system. Recall that S_j receives all messages delivered in the system after receiving the view change message that informs about its join (virtual synchrony). In this case, S_p would only need to send all WSs received before the view change message. The synchronization message MSG_SYNC would not be necessary. Instead, S_j would simply execute all WSs in the buffer as remote transaction after the data transfer from S_p is completed. However, since the recovery can take a long time, it might be that S_j has to buffer many messages. Thus, this strategy requires to implement a persistent buffer because not all buffered WSs might fit into main memory. In contrast, we wanted to make sure that S_j has to buffer only few WSs. In all tests conducted in our implementation, only one WS is buffered in $WSBuffer$ and needs to be processed if no more than 10 WS are sent in one single message from S_p .

Figure 2 shows the actions at the peer site. After agreeing to help with recovery (line 3-7), it sends WSs in several rounds (line 8-15). It first appends all WSs that have to be

```

1.  $S_j$  starts up
2. Performs Local Recovery
3.  $START\_BUFFER := false$ 
4.  $WSBuffer := emptylist$ 
5. Joins the communication group
6. Retrieves  $maxGid$  from WSL
7.  $lastGid := maxGid$ 
8. For each potential peer  $S_p$  {
9.    $send(S_p, MSG\_REQUEST)$ 
10.   $recv(S_p, type)$ 
11.  IF  $type = MSG\_APPROVE$ 
12.     $send(S_p, MSG\_LAST\_TXN, lastGid)$ 
13.  BREAK;
14. ELSE
15.   //  $type = MSG\_DENY$ 
16.   IF all potential peers have been tried
17.     System exists with recovery failure
18. }
19. FOR (; ;) {
20.  Upon connection request from a client
21.   Declines the request;
22.  Upon  $mcast\_recv(MSG\_WS, WS)$  from GCS
23.   IF  $START\_BUFFER = false$ 
24.     Drops WS
25.   ELSE
26.     Appends WS to  $WSBuffer$ 
27.  Upon  $recv(S_p, MSG\_UPD, NWS)$ 
28.   For each WS in  $NWS$ :  $apply(WS)$ 
29.    $send(S_p, MSG\_CONTINUE)$ 
30.  Upon  $mcast\_recv(MSG\_SYNC, NIL)$ 
31.   IF the message was multicast by  $S_p$ 
32.      $START\_BUFFER := true$ 
33.  Upon  $recv(S_p, MSG\_REC\_DONE, NWS)$ 
34.   For each WS in  $NWS$ :  $applyWS(WS)$ 
35.   Blocks GCS channel
36.   For each WS in  $WSBuffer$ 
37.      $applyWS(WS)$ 
38.   Unblocks GCS channel
39.   // Recovery is done; accept client requests
40.   BREAK;
41. }

```

Figure 1: PCS recovery at joining site S_j

transferred to a $WSList$ (lines 16-24). Then it takes WSs from $WSList$ and sends them in sets of N messages to S_j (lines 27-28). Since new WSs might be added to WSL during this process, $WSList$ is updated accordingly whenever needed (lines 12-13). Once $WSList$ contains less than N messages and there are also no new messages in WSL , the synchronization point is found. S_p sends first the synchronization message, waits until it receives it back, and then sends all WSs that were delivered before the synchronization message to S_p (lines 30-36). This completes S_p 's tasks.

4.4.2 TCS algorithm

When TCS is performed, recovery first reads and transfers

```

1. new  $WSList()$ 
2. FOR (; ;) {
3.  Upon  $recv(S_j, MSG\_REQUEST)$ 
4.   IF assisting another site for recovery
5.      $send(S_j, MSG\_DENY)$ 
6.   ELSE
7.      $send(S_j, MSG\_APPROVE)$ 
8.  Upon  $recv(S_j, MSG\_LAST\_TXN, gid)$ 
9.    $buildWSList(gid, -1)$ 
10.   $sendWS()$ 
11.  Upon  $recv(S_j, MSG\_CONTINUE)$ 
12.   IF  $WSList.size \leq N$ 
13.      $buildWSList(0, -1)$ ;
14.    $sendWS()$ 
15. }

16.  $buildWSList(fromGid, toGid)$ 
17.  IF  $fromGid = 0$ 
18.    $maxGid := \max \text{GID in } WSList$ 
19.  ELSE
20.    $maxGid := fromGid$ 
21.  IF  $toGid = -1$ 
22.   Appends WSs where  $GID > maxGid$  to  $WSList$ 
23.  ELSE
24.   Appends WSs where  $toGid \geq GID > maxGid$  to  $WSList$ 

25.  $sendWS()$ 
26.  IF  $WSList.size > N$ 
27.   Moves the first  $N$  WS from  $WSList$  to  $NWS$ 
28.    $send(S_j, MSG\_TXN\_UPD, NWS)$ 
29.  ELSE
30.    $mcast(MSG\_SYNC, NIL)$ 
31.    $mcast\_recv(MSG\_SYNC, NIL)$ 
32.    $upToGid := GetCurrGID()$ 
33.    $buildWSList(0, upToGid)$ 
34.   Moves all WS from  $WSList$  to  $NWS$ 
35.    $send(S_j, MSG\_REC\_DONE, NWS)$ 
36.    $endAssistance()$ 

```

Figure 2: PCS Recovery at the peer site S_p

a snapshot of the database and then performs PCS transferring the remaining transactions. For the peer site, we can insert the code displayed in Figure 3 into Figure 2 between line 8 and 9. The peer site first decides whether to use PCS or TCS (lines 2-5). If the joining site has missed more than $N_{threshold}$ update transactions, TCS will be used. TCS must also be used if S_j is a new site. Recall that when a new site joins the system, its WSL is empty. When such site retrieves $maxGid$ from its WSL (in Figure 1 at line 5), $maxGid$ is assigned -1 . Then, the joining site sends -1 as the $lastGid$ in MSG_LAST_TXN message. When S_p receives it and tries to find it from its WSL (in Figure 3 at line 5), it cannot be found. Hence TCS will be used. Finally, if S_j has joined via TCS after S_p failed, we also have

```

1. //Upon receipt ( $S_j, MSG\_LAST\_TXN, gid$ )
2. //Determines which recovery strategy to use
3.  $maxGid$  = the max GID in WSL
4.  $diff := maxGid - gid$ 
5. IF ( $diff > Nthreshold$ 
   OR  $gid$  cannot be found in WSL
   OR  $lastTCSrecovery > gid$ )
6. // Uses TCS
7. Starts snapshot transaction TS
8.  $toGid :=$  GID of last txn committed before TS starts
9.  $send(S_j, MSG\_USE\_TCS, toGid)$ 
10. Waits until TS completes and saves snapshot to a file
11.  $endAssistance();$ 
12. ELSE
13. // Uses PCS

```

Figure 3: Extension for the peer site

```

1. Upon receipt ( $S_p, MSG\_USE\_TCS, lastGid$ )
2. Writes  $lastGid$  to a file
3. Prints to screen that TCS is used
4. System exits

```

Figure 4: Extension 1 for the joining site

to use TCS. For that, each site keeps track in the variable $lastTCSrecovery$ the GID of the last transaction that was included in the transferred snapshot when it joined the system via TCS. If S_j has $lastTCSrecovery > gid$ then S_j 's own TCS recovery took place after S_j crashed. Therefore, S_j has also to recover via TCS.

If TCS is used, S_p takes a snapshot as described above. It then informs S_j about the last transaction included in this snapshot. Then the recovery is finished for the peer (line 6-11).

For the joining site, the code displayed in Figure 4 needs to be added to Figure 1 between lined 19 and 20. When the joining site receives the MSG_USE_TCS message, it stores the gid in a file, signals on the screen that TCS needs to be used, and then shuts itself down. The system administrator's help is now needed. When the system administrator gets the signal, he/she waits until the snapshot is taken at the peer site, copies it to the joining site, restarts S_j in non-replication mode and installs the snapshot. S_j may not run in replication mode because applying the snapshot may not lead to writesets sent to the other replicas. After the snapshot installation is completed, the administrator shuts down S_j and restarts it in replication mode. When now S_j again restarts, it must know whether a snapshot was installed. For that we have to also replace lines 6 and 7 of Figure 1 with the code in Figure 5.

4.4.3 Failures

The algorithm can be easily extended to handle crashes during recovery. If the peer site fails at any time during the recovery, the joining site terminates the recovery and exits. This will require the system administrator to restart the site again. In case of PCS, the WSLs that have already been transferred to the joining site and applied do not need be

```

1. IF  $lastGid$  is specified in the configuration file
2. Retrieves  $lastGid$  from the file
3.  $lastTCSrecovery := lastGid$ 
4. ELSE
5. Retrieves  $maxGid$  from WSL
6.  $lastGid := maxGid$ 

```

Figure 5: Extension 2 for the joining site

retransferred during the next round of recovery. Similarly, if the joining site itself crashes during the recovery, the system administrator just starts the site again.

4.4.4 Optimized PCS

In some cases, PCS can be extended to further reduce recovery time. The idea is that for each data item that was updated during the downtime of the recovering site S_j , S_j only applies the final version of the data item. For instance, one could scan in reverse order through all relevant writesets and take for each data item only the latest version and apply it at S_j . Applying all intermediate version, as done by PCS, could be avoided. This scanning could be done by the peer site S_p who then only sends the last data versions to S_j . Alternatively, S_p would send all relevant writesets and then S_j performs the scanning. The first solution has the advantage that less messages are sent. The second has the advantage that S_j receives all writesets – an important pre-requisite for S_j to become a peer in future PCS. Furthermore, the burden of scanning and analyzing is not put on S_p . As S_p concurrently serves client requests, it is desirable to keep the recovery overhead at S_p low. In contrast S_j is not yet serving any clients and thus is likely to have more available resources.

The optimized PCS, however, can increase the complexity of the implementation considerably. A first problem arises, if a writeset does not contain the entire modified data item. In some circumstances, a writeset might contain for an updated record only the new values of the attributes that were changed by the update. For instance, if the writeset is extracted from the redo log, this is likely the case. If the writeset only contains the updated attributes, one has to assemble different writesets in order to capture for a given data item the updates on all attributes. This can be quite complicated. A second disadvantage is that the existing functionality of applying writesets cannot be reused. Postgres-R (and other replication solutions) applies writesets on remote sites during normal processing and PCS can simply reuse this functionality. This also helps in not restarting recovery from scratch if the peer site fails during the recovery procedure (see above). In contrast, using optimized PCS, a completely new function needs to be implemented to apply all final versions of data items.

4.5 Implementation Details

Postgre-R itself is an extension of the open-source database management system PostgreSQL. As depicted in Figure 6, a Postgres-R system is a cluster of nodes, and each node is composed of the following processes: *postmaster*, *local backend*, *replication manager*, *replication backend* and *communication manager*. The group communication system, in this

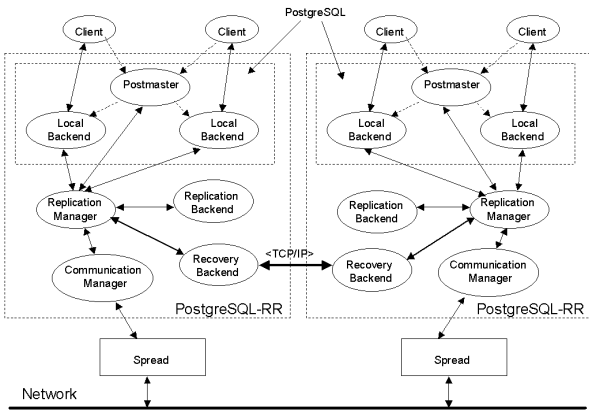


Figure 6: Architecture of Postgres-R

case Spread [25], is not part of the system but used for membership services and total order, uniform reliable multicast. Postmaster and local backends are the only processes that exist in a non-replicated system. Local clients connect to the postmaster that creates a new local backend specifically for this client. A local backend accepts SQL statements from its client, executes them locally and generates the writeset. The replication backend is responsible to apply writesets of remote transactions. The replication manager coordinates the work of the other processes and the communication between them. The communication manager simply provides an abstraction of the GCS to the replication manager.

For distributed recovery, we added a new component, the *recovery backend*. At the peer site, it is responsible to retrieve the WSs from the WSL for PCS or to execute the snapshot transaction for TCS. At the joining site it is responsible for applying the WSs in PCS. The replication manager also has some tasks during PCS recovery. It forwards the *MSG_SYNC* message through the GCS, determines *upToGid* and maintains the *WS_Buffer* list.

Writeset Log We were not able to reconstruct writesets from PostgreSQL’s write-ahead-log that had been developed for local recovery. This was due because the system sometimes writes entire pages instead of only after-images of the updated records. Thus, we stored the writesets together with their GIDs in a system catalog as part of the transaction. PostgreSQL uses system catalogs to record meta-data of the database. Access to the system catalog occurs within the boundaries of a transaction. This eases the implementation of the WSL dramatically. Only after a transaction commits does its WS become visible in the system catalog. The system catalog has a column storing the GID and a column storing the serialized writeset as a Binary Large Object. PostgreSQL automatically compresses it before it is stored into the catalog to save storage space. A B-tree index is built on the GID column to support the range queries needed by the *buildWSList()* function to retrieve sets of writesets.

Alternatively to a system catalog, we could have used a standard file where writesets and their GIDs are inserted. Since transactions commit sequentially according to their GID, a simple append file should be sufficient. This would have probably been more efficient during normal processing than using a system catalog. The problem is that it is

difficult to make the append operation transactional. That means, the file might contain writesets of transactions that eventually did not commit. However, if the joining replica, when applying the writesets received from the peer replica, performs validation as it does during normal processing when applying writesets (and it does so in our implementation), then it would be able to determine unsuccessful transactions. We are planning to compare the system catalog solution with the file-based solution in our next version of the system.

Snapshot Transaction To use TCS, we first take a snapshot of the peer, transfer it to the joining site, and then install the snapshot at the joining site. PostgreSQL provides two functions to front-end users: *pgdump* and *pgrestore*. These functions are often used for backup or system migration.

Pgdump starts a read-only transaction and takes a consistent snapshot of the database as discussed before. *Pgrestore* basically converts the snapshot into SQL statements and feeds them to PostgreSQL. *Pgrestore* must be executed with PostgreSQL running in non-replication mode, since no writesets need to be built and be propagated to other running sites. We used the Unix function *scp* to copy the snapshot file from the peer to the joining site.

A disadvantage of *pgdump* and *pgrestore* is that it is considerably slower than simple file copy techniques. On the other hand, if a full copy of the files of the database are made, it would not be possible to run transactions concurrently, and the peer node would need to be shut down for this purpose (and recovered later on).

One challenge was that the joining node must receive together with the version of each data item the GID of the transaction that created this version. This is needed to perform validation for writesets received from the GCS after recovery has completed.

Optimized PCS We did not implement the optimized PCS for complexity reasons. In Postgres-R, a writeset contains for an updated record only the attributes that were changed. The reason is that this reduces the amount of data that is transferred during writeset propagation. It also has been of advantage for us when logging the writesets in the system catalog since it keeps the record sizes at reasonable levels. However, this made it very hard to determine the complete latest version of a given record. Furthermore, in our implementation, the joining site S_j simply reuses the functionality of applying writesets as provided by Postgres-R.

5. EVALUATION

Our evaluation analyzes the cost of the two recovery strategies, and shows how to determine the threshold value to choose between PCS and TCS. Our experiments use the Open Source Development Lab’s Database Test 1 (OSDL-DBT-1) kit [19], which is similar to the TPC-W benchmark [27] and simulates an online bookstore. There are 12 tables that record customer information, book information and order information. The size of the database can be varied by choosing different numbers of customers and items. The workload can be determined by choosing a different mix of browsing (read-only) and ordering (update) transactions. The system throughput is determined by the number of users simulated by the driver and the *thinktime* used by each user.

We ran our experiments in a local area network using two PCs running 2.6.12.2-smp Linux Kernel. Each PC has

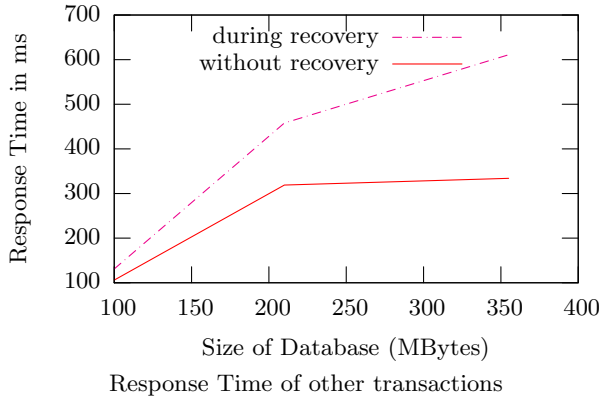
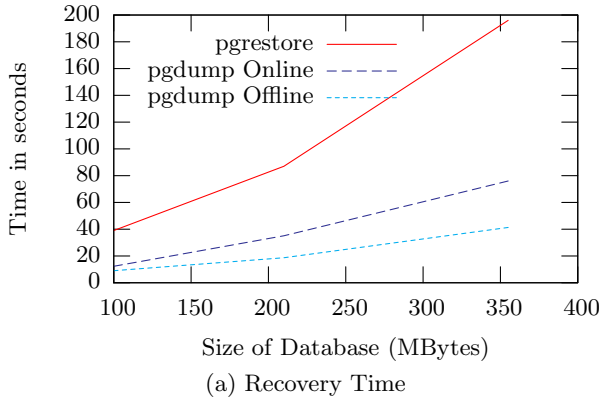


Figure 7: Total Copy Strategy

512MB RAM and two Pentium III CPUs at 733.86 MHz. One machine runs the peer replica, the other the joining replica. We did not use more replicas since they would not be involved in the recovery process. The database is built according to the requirements of the benchmark. We chose the browsing profile where about 80% of the transactions are browsing transactions and the rest are ordering transactions. We chose *thinktime* to be 3.0 seconds. The benchmark driver simulates 100 clients and maintains 20 connections with the peer replica for the clients. The chosen workload kept the replica relatively busy but did not overload it.

5.1 TCS Recovery

The major factor affecting the recovery time of TCS is the size of the database. We tested the overhead for TCS both for offline and for online recovery.

The setup of the experiments is quite straight forward. First, we started a server S_p and generated a database of the desired size. Then we started a server S_j with a configuration file indicating S_p as potential peer site to perform recovery. For online recovery, before S_j was joining the system, we started the clients at S_p . No clients were running on S_p for offline recovery. Since S_j initially had an empty WSL, the TCS strategy was automatically chosen.

Figure 7(a) shows the time needed for executing the snapshot transaction (*pgdump*) at the peer site S_p and applying the snapshot at S_j (*pgrestore*) with increasing database size. For S_p times are given both online (with concurrent clients) and offline (without concurrent clients). Note that at the joining site, no clients should be started until recovery has completed, hence, there is no differentiation between online and offline. The total recovery time is the sum of *pgdump* and *pgrestore* plus the time it takes the administrator to copy the recovery file from S_p to S_j . In this context, the time for the execution of the Unix function *scp* was nearly negligible. The time for *pgdump* at the peer site increases linearly with the database size and is considerably smaller than the time for *pgrestore* at the joining site. Still, one can see that running transactions on the peer node concurrently to creating the snapshot slows execution down since all compete for CPU and I/O. The time for *pgrestore* at the joining site increases faster with increasing database size than the time for *pgdump* at the peer site. This is likely to be true because *pgdump* is a read-only transaction, while *pgrestore* creates update transactions with many inserts.

In order to verify that both *pgdump* and *pgrestore* behave linearly with even larger databases, we conducted similar experiments with a simpler database which was faster to create and where the database size was easier to control. We were able to confirm linear behavior for both *pgdump* and *pgrestore* for databases up to 1 Gigabyte. In fact, the absolute numbers were very similar to the numbers shown for the OSDL-DBT-1 benchmark in Figure 7 and thus, omitted here.

Figure 7(b) shows the response time clients experience when submitting a *Home* transaction (a transaction type of the benchmark) to the peer site, with and without recovery running in the background. Without recovery, the response time first increases with increasing database size up to around 200 MByte and remains relatively stable after that. The response time increases as the hit ratio decreases until the hit ratio stabilizes at around 200 MByte. The stabilization is probably due to the fact that each transaction accesses a certain amount of "hot spot" data that always fits in main memory, while the rest of the accesses is unlikely to be in main memory if the database is larger than 200 MByte. Response times when the peer executes a snapshot transaction concurrently, are considerably higher than when no recovery takes place and the gap increases with increasing database size. With a small database of 100 Mbyte response times without and with recovery are nearly the same, while with a 350 Mbyte database response times during recovery are nearly double as high as when no recovery process is running. This is due to the fact that *pgdump* and client transactions compete for resources and resource contention occurs.

5.2 PCS Recovery

The major factor that affects the recovery time using the PCS approach is the total number of writesets (WS) to be transferred and applied. To run an experiment, we first started site S_p and S_j indicating no recovery is needed. S_p and S_j had identical data. Due to the setup of the benchmark, we only connected clients to S_p . The replica control mechanism guaranteed that all changes were propagated to S_j . After a while, we manually crashed S_j while S_p continued executing client transactions. When the desired number of transactions had been submitted to S_p , we specified S_p as the potential peer in the configuration file and started S_j up again in replication mode. For offline recovery, we

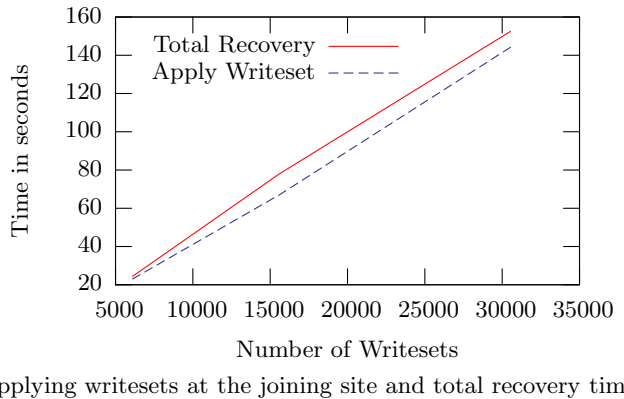
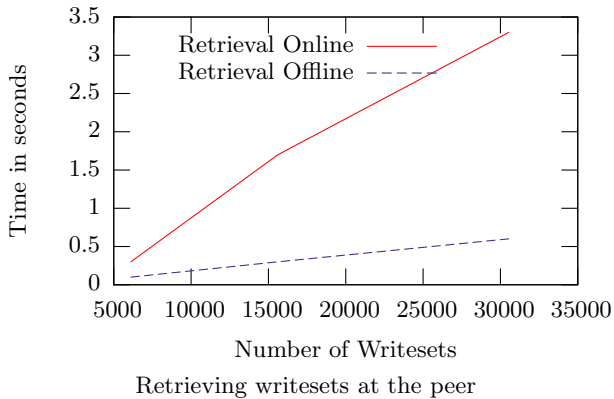


Figure 8: Partial Copy Strategy

stopped the clients at S_p before S_j 's startup. For online recovery, they continued submitting transactions to S_p . Since the WSL of S_j was not empty at startup, the $Nthreshold$ determined which recovery strategy to be used. We set $Nthreshold$ to infinity to force a PCS recovery.

Figure 8 (a) presents the time needed to retrieve all writesets at peer site S_p from WSL that S_j missed. The time is recorded for both online and offline recovery. Retrieval time increases linearly with the number of writesets and is considerably higher for online recovery. This is true, because if no concurrent transactions are running, the snapshot transaction reads from each data item the latest version (which is indexed). In contrast, if concurrent transactions are running, the snapshot transaction has to retrieve older data versions which can take much more time. However, the absolute numbers are only a few seconds, hence, the peer site is very little affected by this form of recovery. In fact, we measured the response time for transactions running on S_p during online recovery, and we could not determine any significant increase compared to the response time when no recovery takes place. Figure 8 (b) presents the time needed to apply the missed writesets at the joining site S_j and the total recovery time for online recovery (the time for offline recovery was only slightly lower). Both times increase linearly with the number of writesets. As the figure indicates, applying the writesets contributes to more than 90% of the total recovery time. The remainder of the time indicates the time needed to retrieve the writesets at the peer, the time for the GID synchronization, the time it takes to transfer WSs over the network, and the time it takes to process the buffered WSs.

An interesting question is whether online recovery is possible from a performance point of view. If during online recovery the remaining system commits more transactions per second than can be transferred with PCS to the joining site, the joining site will never catch up. That is, recovery must be faster than the system commits new transactions. Recovery time per writeset to transmit is around 5 ms in our results for this particular benchmark. That is, 200 writesets can be transferred and applied per second. This means that in our configuration, if the system throughput is more than 200 update transactions per second, recovery is not possible. Note that the throughput of read-only transactions is not relevant and can be arbitrarily high since read-only transactions do not trigger data transfer.

5.3 Determining the Threshold

In order to determine the value of $Nthreshold$ for a specific application the system needs to run example configurations in order to derive the TCS recovery time (dependent on the database size), and the PCS recovery time (dependent on the number of missing transactions) to derive Figures similar to Figures 7 and 8. Only a few test configurations need to be run since the behavior is linear. Then $Nthreshold$ can be calculated dynamically. At the time of recovery, the system determines the database size and the estimated TCS recovery time t_1 for this database size. Then, $Nthreshold$ is set to the number of writesets that can be transferred using the PCS strategy in this given time t_1 . At the same time, the peer site determines the number of writesets that the joining site has missed so far. If it is larger than $Nthreshold$, TCS is faster, otherwise PCS. Note that the calculation does not need to consider the transactions that will be executed after the start of the recovery. Independently of whether TCS or PCS is chosen, they will have to be transferred via PCS.

In our experiment, for a database of 200 MByte TCS recovery time is around 120 seconds. Thus, $Nthreshold$ will be around 25,000 writesets. Given that the total throughput is around 33 transactions per second and the update throughput is around 7 transactions per second in our experiments, if a node is down for more than an hour, TCS will be used, otherwise PCS.

5.4 Other Experiments

We conducted similar experiments with a simpler benchmark. The database contains 20 tables, each having five attributes (two integers, one 40-character, one float and one date). The workload consisted of only update transactions with one to three update operations (each updating one tuple). Throughput was much higher and reached 280 transactions per second. Relative behavior for all experiments was very similar to the OSDL-DBT-1 benchmark. In absolute numbers, the recovery time for TCS was very similar to the recovery time for the OSDL-DBT-1 benchmark, PCS was able to transfer much more writesets per second (550) than in the OSDL-DBT-1 benchmark due to the simpler structure of the transactions.

6. RELATED WORK

Most of the work in cluster replication does not discuss

recovery of failed sites or how new sites can be added to the system. In our previous work on Postgres-R [28], we mention how recovery can be done in principle, but do not present a formal algorithm or a performance evaluation. In [16] we present a series of recovery algorithms, however, no implementation or evaluation is provided. The algorithms presented here differ from [16] due to the fact that PostgreSQL is based on snapshot isolation and not 2-phase-locking. [15] present recovery for middleware-based replication, hence, the solution looks quite different. They also do not provide an evaluation of their approach. [11] presents a middleware based recovery approach that is also based on snapshot isolation. The paper does not explain how the system determines which transactions the recovering site missed, and how recovery is coordinated with transaction processing.

Amza et al.[8, 12] present provisioning mechanisms for replicated databases. The authors assume an infrastructure that hosts several database applications. They then analyze dynamically the incoming load for each of the applications and decide on how many replicas each of the applications will be run. However, all databases are replicated on all machines. The system only decides to which replicas the read-only transactions are redirected.

Quorum replication [26] provides incremental implicit recovery. When a site restarts it receives the latest version of an data item whenever it participates in a write quorum for this data item. Commercial systems (such as Oracle or DB2) mainly implement lazy replication schemes, in which writesets are only propagated some time after transaction commit. These schemes often provide implicit recovery. In here, writeset propagation is often implemented in form of persistent transactional queues. If a site is down the producer of a writeset stores it locally until the receiver restarts upon which propagation is resumed.

Recovery has also been addressed in the context of process replication, such as FT-CORBA [18]. Due to the differences in the system model, state transfer and synchronization are quite different. In the context of reliable multicast, recent approaches have analyzed how to guarantee message delivery in a crash-recovery model [5]. In here, message logging is used to guarantee that a recovering node receives all messages it missed during its downtime.

7. CONCLUSION

This paper presents the design and implementation of a hybrid recovery algorithm for a replicated database cluster. It allows both previously crashed and completely new sites to join a running replicated system without stopping transaction execution in the rest of the system. The algorithm can transfer either the whole database from a running site of the system to the joining site (i.e., the TCS strategy), or only the update information of transactions missed by the joining site during its downtime (i.e., the PCS strategy). The algorithm dynamically chooses the best transfer strategy based on the feasibility and the estimation of the recovery cost.

8. REFERENCES

- [1] C. Amza, A. L. Cox, and W. Zwaenepoel. A comparative evaluation of transparent scaling techniques for dynamic content servers. In *Int. Conf. on Data Engineering (ICDE)*, 2005.
- [2] T. Anderson, Y. Breitbart, H. F. Korth, and A. Wool. Replication, consistency, and practicality: Are these mutually exclusive? In *ACM SIGMOD Conf.*, 1998.
- [3] H. Berenson, P. A. Bernstein, J. Gray, J. Melton, E. J. O’Neil, and P. E. O’Neil. A critique of ANSI SQL isolation levels. In *ACM SIGMOD Conf.*, 1995.
- [4] K. Böhm, T. Grabs, U. Röhm, and H.-J. Schek. Evaluating the coordination overhead of synchronous replica maintenance in a cluster of databases. In *Euro-Par Conf.*, 2000.
- [5] R. Boichat and R. Guerraoui. Reliable and total order broadcast in the crash-recovery model. *J. Parallel Distrib. Comput.*, 65(4), 2005.
- [6] Y. Breitbart, R. Komondoor, R. Rastogi, S. Seshadri, and A. Silberschatz. Update propagation protocols for replicated databases. In *ACM SIGMOD Conf.*, 1999.
- [7] E. Cecchet, J. Marguerite, and W. Zwaenepoel. C-JDBC: Flexible database clustering middleware. In *USENIX Conference*, 2004.
- [8] J. Chen, G. Soundararajan, and C. Amza. Autonomic provisioning of backend databases in dynamic content web servers. In *Int. Conf. on Autonomic Computing (ICAC)*, 2006.
- [9] G. V. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: A comprehensive study. *ACM Computer Surveys*, 33(4), 2001.
- [10] K. Daudjee and K. Salem. Lazy database replication with ordering guarantees. In *Int. Conf. on Data Engineering (ICDE)*, 2004.
- [11] S. Elnikety, S. Dropsho, and F. Pedone. Tashkent: Uniting durability with transaction ordering for high-performance scalable database replication. In *EuroSys*, 2006.
- [12] S. Ghanbari, G. Soundararajan, J. Chen, M. Mihailescu, and C. Amza. Adaptive learning of metric correlations for temperature-aware database provisioning. In *Int. Conf. on Autonomic Computing (ICAC)*, 2007.
- [13] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, 1993.
- [14] J. Holliday, D. Agrawal, and A. E. Abbadi. The performance of database replication with group communication. In *Int. Symp. on Fault-tolerant Computing*, 1999.
- [15] R. Jiménez-Peris, M. Patiño-Martínez, and G. Alonso. Non-intrusive, parallel recovery of replicated data. In *Int. Symp. on Reliable Distributed Systems (SRDS)*, 2002.
- [16] B. Kemme, A. Bartoli, and O. Babaoglu. Online reconfiguration in replicated databases based on group communication. In *Int. Conf. on Dependable Systems and Networks (DSN)*, 2001.
- [17] Y. Lin, B. Kemme, M. Patiño-Martínez, and R. Jiménez-Peris. Middleware based data replication providing snapshot isolation. In *ACM SIGMOD Conf.*, 2005.
- [18] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. State synchronization and recovery for strongly consistent replicated CORBA objects. In *Int. Conf. on Dependable Systems and Networks (DSN)*, 2001.

- [19] Open Source Development Lab. Descriptions and Documentation of OSDL-DBT-1, 2002. <http://sourceforge.net/projects/oslldbt>.
- [20] E. Pacitti, P. Minet, and E. Simon. Replica consistency in lazy master replicated databases. *Distributed and Parallel Databases*, 9(3), 2001.
- [21] E. Pacitti and E. Simon. Update propagation strategies to improve freshness in lazy master replicated databases. *VLDB Journal*, 8(3), 2000.
- [22] F. Pedone, R. Guerraoui, and A. Schiper. Exploiting atomic broadcast in replicated databases. In *Euro-Par Conf.*, 1998.
- [23] C. Plattner and G. Alonso. Ganymed: Scalable replication for transactional web applications. In *Middleware*, 2004.
- [24] U. Röhm, K. Böhm, H.-J. Schek, and H. Schuldt. FAS - a freshness-sensitive coordination middleware for a cluster of olap components. In *Int. Conf. of Very Large Databases (VLDB)*, 2002.
- [25] Spread. homepage: <http://www.spread.org/>.
- [26] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(9), 1979.
- [27] Transaction Processing Performance Council. TPC Benchmark W, 2000. <http://www.tpc.org>.
- [28] S. Wu and B. Kemme. Postgres-R(SI): Combining replica control with concurrency control based on snapshot isolation. In *Int. Conf. on Data Engineering (ICDE)*, 2005.