Distributed Multi-Level Recovery in Main-Memory Databases

Rajeev Rastogi* Philip Bohannon* James Parker* S. Seshadri† Avi Silberschatz* S. Sudarshan†

* Bell Laboratories, Murray Hill, NJ
{rastogi,plbohannon,parker,avi}@bell-labs.com
† Indian Institute of Technology, Bombay, India
{seshadri,sudarsha}@cse.iitb.ernet.in

Abstract

In this paper we present recovery techniques for *distributed main-memory* databases, specifically for client-server and shared-disk architectures. We present a recovery scheme for client-server architectures which is based on shipping log records to the server, and two recovery schemes for shared-disk architectures—one based on page shipping, and the other based on broadcasting of the log of updates. The schemes offer different tradeoffs, based on factors such as update rates.

Our techniques are extensions to a distributed-memory setting of a centralized recovery scheme for main-memory databases, which has been implemented in the Dalí main-memory database system. Our centralized as well as distributed-memory recovery schemes have several attractive features—they support an explicit multi-level recovery abstraction for high concurrency, reduce disk I/O by writing *only* redo log records to disk during normal processing, and use per-transaction redo and undo logs to reduce contention on the system log. Further, the techniques use a fuzzy checkpointing scheme that writes only dirty pages to disk, yet minimally interferes with normal processing—all but one of our recovery schemes do not require updaters to even acquire a latch before updating a page. Our log shipping/broadcasting schemes also support concurrent updates to the same page at different sites.

Appeared in: Distributed and Parallel Databases, Volume 6, Number 1, January 1998, Kluwer Academic Publishers.

†The work of these authors was performed in part while they were at Bell Labs.

1 Introduction

A large number of applications (e.g., call routing and switching in telecommunications, financial applications, automation control) require high performance access to data with response time requirements of the order of a few milliseconds to tens of milliseconds. Traditional disk-based database systems are incapable of meeting the high performance needs of such applications due to the latency of accessing data that is disk-resident. An attractive approach to providing applications with low (and predictable) response times is to load the entire database into main-memory. Databases for such applications are often of the order of tens or hundreds of megabytes, which can easily be supported in main-memory. Further, machines with main memories of 8 gigabytes or more are already available, and with the falling price of RAM, machines with such large main memories will become cheaper and more common.

One approach for implementing such high performance databases is to provide a large buffer-cache to a traditional disk-based system. In contrast, in a *main-memory database system* (MMDB) (see, e.g., [7, 12, 9, 5]), the entire database can be directly mapped into the virtual address space of the process and locked in memory. Data can be accessed either directly by virtual memory pointers, or indirectly via location independent database offsets that can be quickly translated to memory addresses. During data access, there is no need to interact with a buffer manager, either for locating data, or for fetching/pinning buffer pages. Also, objects larger than the system's page size can be stored contiguously, thereby simplifying retrieval or in-place use. Thus, data access using a main-memory database is very fast compared to using disk-based storage managers, even when the disk-based manager has sufficient memory to cache all data pages.

Distributed architectures in which several machines are connected by a fast network, and perform database accesses and updates in parallel, provide significant further performance improvements for a number of applications. For example, consider applications in which transactions are predominantly read-only and update rates are low (e.g., number translation and call routing in telecommunications). Each machine can locally access data cached in memory, thus avoiding network communication which could be fairly expensive. Another example is Computer Aided Design applications, where locality of reference is very high, update transactions are long, and interactive response time is very important.

Distribution also enhances fault tolerance, which is required in many mission-critical applications, even if data fits easily in a single machine's main-memory. In this case, especially with low update rates, a distributed database is preferable to a hot-spare since the load can be distributed in the non-failure case leading to improved performance.

The recovery scheme used in the Dalí main-memory database system [9, 1] is based on the main-memory recovery scheme presented in [10]. The recovery scheme of [10] provides important features such as *transient undo logging* in which undo log records are kept in memory and only written to disk if required for checkpointing, per-transaction logs in memory to reduce contention on the system log tail, and recovery using only a single pass over the system log. The recovery scheme used in Dalí provides several further extensions, such as *multi-level recovery* ([20, 14, 13]), and *fuzzy checkpointing* [18, 8].

The goal of the work described here was to extend the Dalí recovery scheme to the distributed memory case, simultaneously maintaining the advantages of the single-site scheme, and efficiently supporting the applications described above. For example, we can make use of transient undo logging to reduce the size of the log written to disk, as well as the size of the log sent across network links in distributed protocols.

We present three distinct but related distributed recovery schemes – the first for *client-server* architectures, and the second and third for *shared disk* architectures. These are all "data-shipping" schemes (see, e.g., [6]) in which a transaction executes at a single site, fetching data (pages) as required from other sites. Distributed commit protocols are not needed as in "function-shipping" environments. While shared disk architectures have traditionally been closely tied to hardware platforms (e.g., VAXcluster), UNIX-based shared disk platforms and network of workstation architectures with similar performance

characteristics are becoming more common.

A key property of the client-server scheme and one of the shared disk schemes is that concurrent updates are possible at granularities smaller than a page-size, thereby minimizing "false-sharing" (that is, apparent conflicts due to coarse-granularity locking) and consequently, needless network accesses to resolve false sharing. Our distributed recovery algorithms provide the advanced features of our centralized recovery algorithms, such as transient undo logging, explicit multi-level recovery, and fuzzy checkpointing. Site or global recovery requires only a single pass over the system log, starting from the end of the system log recorded in the most recent checkpoint.

The remainder of the paper is organized as follows. We present background on multi-level recovery and the single-site algorithm on which the present work is based in Section 2. Related work is presented in Section 3. We present our client-server recovery algorithm in Section 4. Section 5 describes our shared disk model, while Sections 6 and 7 present our shared disk recovery algorithms. Section 8 concludes the paper.

2 Overview of Main-Memory Recovery

In this section we present a review of multi-level recovery concepts and an overview of the single-site main-memory recovery scheme used in the Dalí system. Low-level details of our scheme are described in [2].

In our scheme, data is logically organized into *regions*. A region can be a tuple, an object, or an arbitrary data structure like a list or a tree. Each region has a single associated lock, referred to as the *region lock*, with exclusive (X) and shared (S) modes that guard updates and accesses to the region, respectively.

2.1 Multi-Level Recovery

Multi-level recovery [20, 14, 13] provides recovery support for enhanced concurrency based on the semantics of operations. Specifically, it permits the use of weaker *operation* locks in place of stronger shared/exclusive region locks.

A common example is index management, where holding physical region locks until transaction commit leads to unacceptably low levels of concurrency. If undo logging has been done physically (e.g. recording exactly which bytes were modified to insert a key into the index) then the transaction management system must ensure that these physical undo descriptions are valid until transaction commit. Since the descriptions refer to byte changes at specific positions, this typically implies that the region locks on the updated index nodes must be held until transaction commit to ensure correct *recovery*, in addition to considerations for concurrent access to the index.

The multi-level recovery approach is to replace these low-level physical undo log records with higher level logical undo log records containing undo descriptions at the operation level. Thus, for an insert operation, physical undo records would be replaced by a logical undo record indicating that the inserted key must be deleted. Once this replacement is made, the region locks may be released and only (less restrictive) operation locks need to be retained. For example, region locks on the particular nodes involved in an insert can be released, while an operation lock on the newly inserted key that prevents the key from being accessed or deleted is held.

2.2 System Overview

Figure 1 gives an overview of the structures used for recovery. The database (a sequence of fixed size pages) is mapped into the address space of each process and is in main memory, with (two) checkpoint

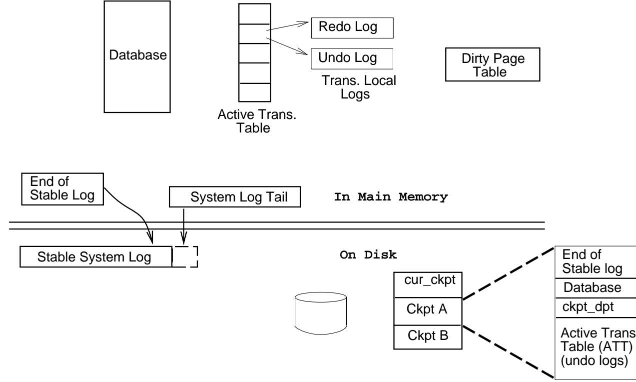

Figure 1: Overview of Recovery Structures

images Ckpt_A and Ckpt_B on disk. Also stored on disk are 1) cur_ckpt, an "anchor" pointing to the most recent valid checkpoint image for the database, and 2) a single system log containing redo information, with its tail in memory. The variable end_of_stable_log stores a pointer into the system log such that all records prior to the pointer are known to have been flushed to the stable system log.

There is a single *active transaction table* (ATT) in main-memory which stores separate redo and undo logs for active transactions, in addition to information about transaction status. A dirty page table, dpt, is maintained in memory to record pages that have been updated since the last checkpoint. For simplicity of presentation, we assume that the dirty page table is maintained as a bitmap with one bit per page. The ATT (with undo logs, but without redo logs) and the dirty page table are also stored with each checkpoint image. The dirty page table in a checkpoint image is referred to as ckpt_dpt.

2.3 Transactions and Operations

Transactions, in our model, consist of a sequence of multi-level operations, similar to [13]. We briefly describe the model below. Each operation has a level L_i associated with it. An operation at level L_i can consist of a sequence of operations at level L_{i-1}. Transactions, assumed to be at level L_n, call operations at level L_{n-1}. Physical updates to regions are level L_0 operations. For transactions, we distinguish between *pre-commit*, when the commit record enters the system log in memory, establishing a point in the serialization order, and *commit* when the commit record hits the stable log. For operations, we use the terms commit and pre-commit interchangeably since both refer to the time when the commit record enters the system log in memory.

Each transaction obtains an *operation* lock before it executes an operation; the operation lock is granted if the operation commutes with other operation locks held by other active transactions. Level L_0 operations obtain region locks instead of operation locks. The locks on the region are released once the L_1 operation pre-commits; similarly, an operation lock at level L_i is held until the transaction or the containing operation (at level L_{i+1}) commits. All the locks acquired by a transaction are released once it commits.[1]

2.4 Logging Model

The recovery algorithm maintains separate *local* undo and redo logs in memory for each transaction. These are stored as a linked list off an entry for the transaction in the ATT. Each physical update (to a

[1] It is possible to release locks for a transaction on pre-commit; as a result read-only transactions may read uncommitted data, and their commit must be delayed until the dirty data they have read has been committed.

part of a region) generates physical undo and redo log records that are appended to the respective local log. When a transaction/operation pre-commits, the current contents of the transaction's local redo log are appended to the system log tail in memory, and the logical undo description for the operation is included in an operation commit log record appended to the system log. Thus, with the exception of logical undo descriptors, only redo records are written to the system log during normal processing.

Also, when an operation pre-commits, the undo log records for its suboperations/updates are replaced in the transaction's (local) undo log with a logical undo log record containing the undo description for the operation. In-memory undo logs of transactions that have committed are deleted since they are not required again.[2]

The system log is flushed to disk when a transaction commits. For each redo log record written to disk, pages touched by the update on the log record are marked dirty in the dirty page table, `dpt`, by the flushing procedure. In our single-site recovery scheme, update actions do not obtain latches on pages – instead region locks are obtained to ensure that updates do not interfere with each other.[3] Eliminating latching significantly decreases access costs in main-memory, and reduces programming complexity. Recovery related actions that are normally taken on page latching, such as setting of dirty bits for the page, are now performed based on log records written to the redo log. (Our distributed-memory schemes, with the exception of one of the shared-disk schemes, do not obtain page latches either; the sole exception uses page latching to ensure cache coherency, which is not a problem in the single-site case.) The redo log is used as a single unifying resource to coordinate the application's interaction with the recovery system, and this approach has proven very useful.

2.5 Ping-pong Checkpointing

Consistent with the terminology in main-memory database literature, we use the term *checkpoint* to mean a copy of the main-memory database which is stored on disk, and the term *checkpointing* to refer to the action of creating a checkpoint. This terminology differs slightly from the terminology used, for example, in ARIES [14].

Traditional recovery schemes implement *write-ahead logging* (WAL), whereby all undo logs for updates on a page are flushed to disk before the page is flushed to disk. In such systems, to guarantee the WAL property, typically a latch on a page is obtained, all log records pertaining to the page are flushed to stable storage, the page is copied to disk, and the latch released. Updaters also obtain the same page latch, thereby preventing concurrent updates while a page is being flushed to disk. As a result of not obtaining latches on pages during updates, it is not possible to enforce the write-ahead logging policy, since pages may be updated even as they are being written out.

Instead, our recovery algorithm makes use of a strategy called *ping-pong checkpointing* (see, e.g., [19]). In ping-pong checkpointing two copies of the database image are stored on disk, and alternate checkpoints write dirty pages to alternate copies. Writing alternate checkpoints to alternate copies permits a checkpoint that is being created to be temporarily inconsistent; i.e., updates may have been written out without corresponding undo records having been written. However, after writing out dirty pages, sufficient redo and undo log information is written out to bring the checkpoint to a consistent state. Even if a failure occurs while creating one checkpoint, the other checkpoint is still consistent and can be used for recovery.

Keeping two copies of a main-memory database on disk for ping-pong checkpointing does not have a very high space penalty, since disk space is much cheaper than main-memory. Further, ping-pong checkpointing has several other benefits. For instance, although many recovery schemes assume page

[2]The logs can be deleted on pre-commit, since, short of a system crash, nothing can result in the transaction aborting.

[3]In cases when region sizes change, certain additional region locks on storage allocation structures may need to be obtained. For example, in a page based system, if an update causes the size of a tuple to change, then in addition to a region lock on the tuple, an X mode region lock on the storage allocation structures on the page must be obtained.

writes are atomic, in reality they are not, and complex schemes are needed to detect and recover from incomplete page writes resulting from, for example, power failures. Incomplete page writes cause no problems with ping-pong checkpointing, since the previous checkpoint image is still available. Ping-pong checkpointing also permits some physical and logical consistency checks to be performed on the checkpoint before declaring it successfully completed.

Before writing any dirty data to disk, the checkpoint notes the current end of the stable log in the variable end_of_stable_log, which will be stored with the checkpoint. This is the start point for scanning the system log when recovering from a crash using this checkpoint. Next, the contents of the (in-memory) ckpt_dpt are set to those of the dpt and the dpt is zeroed (noting of end_of_stable_log and zeroing of dpt are done atomically with respect to flushing). The pages written out are the pages that were either dirty in the ckpt_dpt of the last completed checkpoint, or dirty in the current (in-memory) ckpt_dpt, or in both. In other words, all pages are written out that were modified since the current checkpoint image was previously written, namely, pages that were dirtied since the last-but-one checkpoint. This is necessary to ensure that updates described by log records preceding the current checkpoint's end_of_stable_log have made it in the database image in the current checkpoint.

Checkpoints write out dirty pages without obtaining any latches and thereby avoid interfering with normal operations. The checkpoint image is thus *fuzzy*. Fuzzy checkpointing however could result in two problems for recovery:

- the checkpoint page image may contain partial updates of an operation
- the undo log record for an update may not be in the stable system log (which could result in a problem if the system were to crash immediately after the checkpoint).

The first problem is solved by our policy of always writing physical redo log records. By applying physical redo log records (whose effects are idempotent) to a checkpoint page image we can ensure that we can obtain a page image that does not contain any partial updates.

The second problem is solved by ensuring that for any update whose effects have made it to the checkpoint image, one of the following holds: 1) corresponding physical undo log records are written out to disk after the database image has been written or 2) all physical redo log records for the operation (corresponding to the partial update) as well as the logical undo descriptor in the operation commit log record are on stable storage. This is performed by checkpointing the ATT and flushing the log after checkpointing the data. The checkpoint of the ATT writes out undo log records, as well as some other status information. In case the operation containing the partial update completes and consequently the undo log records are removed from the ATT before the checkpoint of the ATT, the log flush ensures that all log records corresponding to the operation (containing the partial update) as well as the operation commit log record are on stable storage. The checkpoint is declared completed (and consistent) by toggling cur_ckpt to point to the new checkpoint.

2.6 Abort Processing

When a transaction aborts, that is, does not successfully complete execution, updates/operations described by log records in the transaction's undo log are undone by traversing the undo log backwards from the end. Transaction abort is carried out by executing, in reverse order, every undo record just as if the execution were part of the transaction.

Following the philosophy of *repeating history* [14], new physical redo log records are created for each physical undo record encountered during the abort. Similarly, for each logical undo record encountered, a new "compensation" or "proxy" operation is executed based on the undo description. Log records for updates performed by the operation are generated as during normal processing. Furthermore, when the proxy operation commits, all its undo log records are deleted along with the logical undo record for the

operation that was undone. The commit record for the proxy operation serves a purpose similar to that served by *compensation log records* (CLRs) in ARIES – during restart recovery, when it is encountered, the logical undo log record for the operation that was undone is deleted from the transaction's undo log, thus preventing it from being undone again.

2.7 Recovery

Restart recovery begins by initializing the ATT and transaction undo logs to the ATT and undo logs stored in the most recent checkpoint, loads the database image and sets dpt to zero. Next, recovery processes redo log records. Recall that as part of the checkpoint operation, the end of the system log on disk, end_of_stable_log, is noted before the database image is checkpointed. This value of end_of_stable_log becomes the "begin recovery point" for the checkpoint once the checkpoint has completed. All updates described by log records preceding this point are guaranteed to be reflected in the checkpointed database image.

Thus, during restart recovery only redo log records following the end_of_stable_log for the last completed checkpoint of the database are applied. Restart recovery ignores redo log records for updates performed by an operation if the commit log record for the operation is not found in the system log. Such log records represent uncommitted updates, and may not have corresponding undo records in the checkpointed ATT. However, if the undo records are absent, the effects of the log records will not be reflected in the checkpointed database image. Such records would be present only due to a crash while the log records for an operation were being flushed.

During the application of redo log records, appropriate pages in dpt are set to dirty for each log record and necessary actions are taken to keep the checkpointed image of the ATT consistent with the log as it is applied. These actions on the ATT mirror the actions taken during normal processing. For example, when an operation commit log record is encountered, lower level log records in the transaction's undo log for the operation are replaced by a higher level undo description.

Once all the redo log records have been applied, the active transactions are rolled back. To do this, all completed operations that have been invoked directly by the transaction, or have been directly invoked by an incomplete operation, have to be rolled back. However, the order in which operations of different transactions are rolled back is very important, so that an undo at level L_i sees data structures that are consistent [13]. First, all operations (across all transactions) at L_0 that must be rolled back are rolled back, followed by all operations at level L_1, then L_2 and so on.

3 Connection to Related Work

Multi-level recovery and variants thereof, primarily for disk-based systems, have been proposed in the literature [20, 13, 14]. Like these schemes, our schemes repeat history, generate log records during undo processing and log operation commits when undo operations complete (similar to CLRs described in [14]). Also, as in [13], transaction rollback at crash recovery is performed level by level. Some of the features of our main-memory recovery technique which impact the distributed schemes are:

1. Due to transient undo logging, no physical undo logs are written out to the global log except during checkpoints.

2. Separate undo logs are maintained in memory for active transactions. A result is that transaction rollback does not need to access the global log, part of which could be on disk.

3. Our single-site scheme does not require latching of pages during updates, which is inconvenient and expensive in either a main-memory DB or an OODB setting. Actions that are normally taken on page latching, such as setting of dirty bits for the page, are efficiently performed based on physical

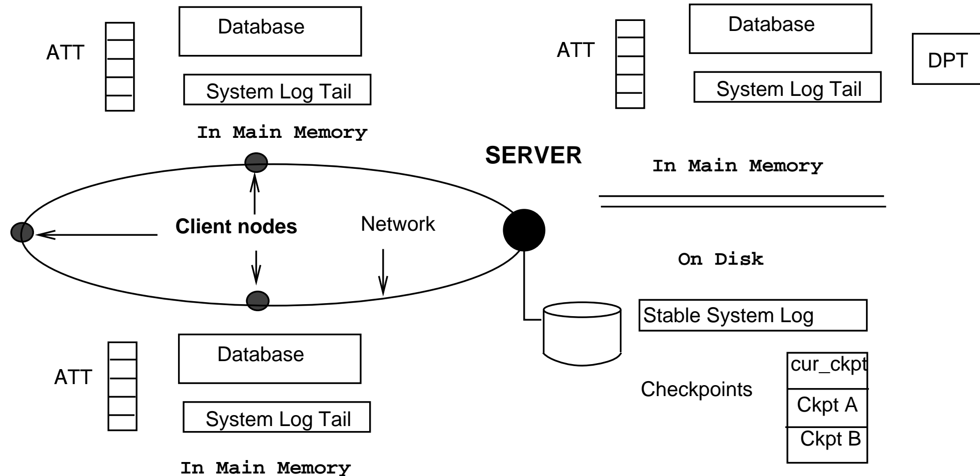

Figure 2: Client-Server Architecture

redo log records written to the global log. (One of our shared-disk schemes uses page latching for ensuring cache consistency, while the other shared-disk scheme does not.)

4. The correctness requirements of the *write-ahead logging* policy are accomplished with a single flush for the entire database during a checkpoint, rather than (potentially) one flush per page.

5. Our scheme does not perform in-place update of the disk image during page flush, instead using ping-pong checkpointing.

In the ARIES-SD [15] family of schemes for recovery in the shared disk environment, each site maintains a separate log, and pages are shipped between sites. Our shared-disk log-shipping scheme does not ship pages, but instead broadcasts log records, taking advantage of cheap application of these log records in main-memory, and permitting *concurrent updates* at a smaller-than-page granularity. In our shared disk schemes, log flushes are driven by the release of a lock from a site, in order to support repeating of history and correct rollback of multi-level actions during crash recovery. The "super fast" method of ARIES-SD [15] does not describe flushes to protect the early release of locks, making it unclear how that scheme supports logical undo and high-concurrency index operations.

In [17], the authors propose recovery schemes for the shared disk environment which assume page-level concurrency control and the NO-STEAL page write policy – neither of which are assumptions made in our schemes.

In [16], the authors show how the ARIES recovery algorithm described in [14] can be extended to a client-server environment. In contrast to our client-server scheme, their scheme involves the clients as well as the server in the checkpointing process. We also support concurrent updates to a page by different clients, which is not supported in [16].

In [4], object-level as well as adaptive locking and replica management are discussed, but recovery considerations are not extensively addressed. In [6], the client-server recovery scheme for the Exodus storage manager (ESM-CS) is described. This recovery scheme, based on ARIES [14], requires page-level locking until end of transaction (for example, the Commit Dirty Page List).

4 Client-Server Recovery Scheme

In this section, we describe the client-server recovery scheme. Our system model is as follows.

- There is a single server with stable storage, which is responsible for co-ordinating all the logging, and for performing checkpoints and recovery (see Figure 2). The server maintains a copy of the entire database in memory.
- Multiple clients may be connected to the server; each client has a copy of the entire database in its memory.
- A transaction executes at a single client and updates/accesses the copy of the database at the client.
- The network is FIFO and reliable.

As a result of updating the local copy of the database, database pages updated by a client may not be *current* at some other client. Therefore, a page at a client is in one of two states – *valid* or *invalid*. Invalid pages contain stale versions of certain data due to updates by other clients and are refreshed by obtaining the latest copy of the page from the server.

Transactions follow the *callback locking* scheme [11, 4] when obtaining and releasing locks. Each client site has a *local lock manager* (LLM) which caches locks and a *global lock manager* (GLM) at the server keeps track of locks cached at the various clients. Transaction requests for locks cached locally are handled at the client itself. However, requests for locks not cached locally are forwarded to the GLM which *calls back* the lock from other clients that may have cached the lock in a conflicting mode (before granting the lock request). A client relinquishes a lock in response to a callback as soon as transactions currently holding the lock (if any) release the lock.

The server maintains the `dpt` and the ATT (for all transactions in the client-server system) while the clients maintain the ATT for the transactions belonging to that client. The log records for updates generated by a transaction at a client site are stored in that site's ATT. Client sites do not maintain a system log on disk, but keep a system log tail in memory and append log records from the local redo logs to this tail when operations commit/abort. Checkpointing is performed solely at the server, and follows the same procedure as the centralized case.

When a lock is relinquished from a site or a transaction commits, log records in the system log are shipped by the client to the server. In the case of transaction commit, the client waits for the server to flush the newly received log records to disk before reporting the commit to the user. The shipped redo log records are used to update the server's copy of the affected pages, ensuring that pages shipped to clients from the server are current (note that pages are shipped only from the server to clients and never vice versa). This enables our scheme to support concurrent updates to a single page at multiple clients since re-applying the updates at the server causes them to be merged (this approach is also adopted in [3]). Shipping the log records will usually be cheaper than shipping pages, and the cost of applying the log records themselves is small since, in our main-memory database context, the server will not have to read the affected pages from disk.

We will now describe our scheme in detail and also outline several possible optimizations to the basic ideas discussed above.

4.1 Basic Operations

We now describe the features which distinguish the client-server scheme from the centralized case, in terms of actions performed at the client and the server at specific points in processing.

- **Page Access:** In case a client accesses a page that is valid, it simply goes ahead without communicating with the server. Else, if the page is *invalid* (certain data on the page may be stale), then the client refreshes the page by 1) obtaining the most recent version of the page from the server, and 2) applying to the newly received page any local updates which have not been sent to

the server (this step merges local updates with updates from other sites). The client then marks the page as valid. The server keeps track of clients that have the page in a valid state.

To prevent race conditions, the client does not send log records to the server after asking for a page and before receiving it.

An optimization of the above is to check for validity of pages at the time of acquisition of region locks from the server rather than on every access; for this optimization to be used, the set of pages covered by the region lock must be known.

- **Operation/Transaction Commit:** At the client, redo log records are moved to the system log, a commit record is appended, and appropriate actions are performed on the transaction's undo log in the ATT as described for the centralized case. In case of transaction commit, the log records in the system log are shipped to the server, and commit processing waits until the server has acknowledged that the log records have been flushed to disk.

  Finally, all the locks acquired by the operation/transaction are released locally. The local lock manager at the site may however continue to cache the locks locally.

- **Lock Release:** When a lock is relinquished by a client, all redo log records that were generated under this lock need to be shipped to the server. The server then applies these log records to its database image to ensure that another client that obtains the same lock gets a copy of the pages which contains the updates described by these log records. A simple way to ensure that all log records generated under the lock are shipped to the server is to flush the system log from the client to the server.

  An optimization to avoid flushing the system log each time is to store the end of the client system log with the lock (at the client) when a X mode region lock or an operation lock is released by a transaction. Thus, for any region lock, all redo log records in the system log affecting that region precede the point in the log stored with the lock. Similarly, for an operation lock, all log records relating to the operation (including operation commit) precede the point in the system log stored with the lock. This location in the log is client-site-specific.

  Before a client site relinquishes an X mode region lock or operation lock to the server due to call-back, it ships to the server at least the portion of the system log which precedes the log pointer stored with the lock. This ensures that the next lock will not be acquired on the region until the server's copy is up to date, and the history of the update is in place in the server's logs. For X mode region locks, this flush ensures repeating of history on regions, while for operation locks this flush ensures that the server receives the logical undo descriptors in the operation commit log records for the operation which released the locks. Thus, if the server aborts a transaction after a site failure, the abort of this operation will take place at the logical level of the locks still held for it at the server.

- **Log Record Processing:** At the server, for each physical redo log record (received from a client), the undo log record is generated by reading the current contents of the page at the server. The new log record is then appended to the undo log for this transaction in the server's ATT. Next the update described by the redo log record is applied, following which the log record is appended to the redo log for the transaction in the server's ATT. Operation/transaction commit log records received from the client are processed by performing the same actions as in the centralized case when the log records were generated. In addition, for operation commit, the logical undo descriptor is extracted from the commit log record and appended to the undo log for the transaction in the server's ATT. For transaction commit, the client whose transaction committed is notified after the log flush to disk succeeds.

By applying all the physical updates described in the physical log records to its pages, the server ensures that it always contains the latest updates on regions for locks which have been released to it from the clients. The effect of the logging scheme, as far as data updates are concerned, is just as if the client transaction actually ran at the server site.

- **Transaction Abort/Site Failures:** If a client site decides to abort a transaction, it processes the abort (as in the centralized case) using the undo logs for the transaction in the client's ATT. If the client site itself fails, the server will abort transactions that were active at the client using undo logs for the transaction in it's ATT(since the client cannot commit without communicating with the server, in case of partition, a decision to abort is enforceable by the server). If the server fails, then the complete system is brought down, and restart recovery is performed at the server as described in Section 2.7.

- **Page Invalidation**

  We complete our client-server scheme by presenting two methods, invalidate-on-update, and invalidate-on-lock, for ensuring that data accessed by a client is up-to-date.

  All actions described so far are used in common by both methods. In particular, both methods follow the rule that all log records pertaining to updates made under a lock are flushed to the server before the lock is relinquished from the site. Since the server would have applied the log records to its copy of the data, this ensures that when the server grants a lock, it has the current version of all pages containing data covered by that lock. However, when a client acquires a lock, it is still possible that the copy of one or more pages involved in the region for which the lock was obtained are not up-to-date at the client.

  Both methods mark pages at the clients as invalid, to denote that some of the data on the page is out of date. Even if a page is marked invalid, some of the data in the page may still be up-to-date, for instance, if the client has a region lock on the data. The first method, invalidate-on-update, is an eager method that marks pages as invalid at clients as soon as an update occurs at the server, while the second, invalidate-on-lock, is a more lazy method, marking pages as invalid at clients when the client gets a lock. The second scheme reduces invalidation messages by keeping extra per-lock information at the server. Details of the two methods are presented in Sections 4.2 and 4.3 respectively.

4.2 Invalidate-On-Update

The invalidate-on-update scheme works as follows. When the server receives log records from a client, it does the following. For each page that it updates, it sends *invalidate* messages to clients (other than the client that updated the page) that may have the page marked as valid. For all clients other than the client that updated the page, the server notes that the client does not have the page marked valid. Clients, on receiving the invalidate message, mark their page as invalid. Thus invalidation messages are received by clients before they can acquire a region lock on the updated data, and begin accessing the data.

Although the method is very simple and easy to implement, it has some drawbacks. For example, consider two sites s_1 and s_2 updating the same page concurrently under two different region locks. Let s_1 be the site that flushes its updates to the server first; the update will cause the server to send an invalidate message to s_2, which will then re-read the page from the server. However, if site s_2 accesses the page again *under the lock that it already has*, then the invalidate was not necessary, since the data in the region it has locked has not changed. The invalidate-on-lock scheme in the next section takes advantage of this observation to reduce overheads.

4.3 Invalidate-On-Lock

The invalidate-on-lock scheme decreases unnecessary invalidations and the overhead of sending invalidation messages by marking pages as invalid only when a lock on a region covering the page is obtained by a client. As a result, if two clients are updating different regions on the same page, as in the earlier example, no invalidation messages are sent to either client. By piggy-backing invalidation messages for updated pages on lock grant messages from the server, the overhead of sending separate invalidation messages in the previous scheme is eliminated.

The biggest benefit of the invalidate-on-lock scheme, however, is that there is no need to check for validity of a page on every access or update to the page—it suffices to check for validity at lock acquisition time.

To achieve the above, the scheme must associate with the lock for a region information about updates to that region. Specifically, when updates described by a physical redo record are applied to pages at the server, the updated pages are associated with the lock for the updated region. Thus, the scheme requires that it be possible to determine the region lock from the redo record. A simple way of obtaining this information is to require that an update call must specify not only the data to be updated, but also the region lock that protects the data. It is easy for a programmer to provide this information, since all updates must be made holding a region lock. The lock name can then be sent with the redo log record.

This scheme also requires that the server associate a *Log Sequence Number* (LSN), with each log record, which reflects both the order in which the record was applied to the server's copy of the page and the order in which it was added to the system log. For each page, the server stores the LSN of the most recent log record that updated the page, and the identity of the client which issued it. In addition, for each client, the server maintains in a *client page table* (cpt), the state of the page at the client (valid/invalid), along with the LSN for the page when it was last shipped to the client.

The server also maintains for each region lock a list of pages that are dirty due to updates to the region. For each page in the list, the server stores the LSN of the most recent log record received by the server that recorded an update to the part of the region on this page, and the client which performed the update. Thus, when a client is granted a region lock, if, for a page in the lock list, the LSN is greater than the LSN for the page when it was last shipped to the client, then the client page contains stale data for the region and must be invalidated.

The LSN information serves to minimize the shipping of pages to clients, marking a page as invalid only if there is an update performed under the region lock requested by the client, and the update has not yet been propagated to the client.

The additional actions for this scheme are as follows:

- **Log apply:** When the server applies to a page P a redo log record, LR, generated at client C under region lock L, it takes the following actions (after P has been updated). First, the LSN for P is set to the LSN for LR. Second, the entry for P in the list of dirty pages for L is updated (or created), setting the client to C, and the LSN to the LSN for LR.

- **Lock grant:** A set of invalidate messages is passed back to the client with the lock acquisition. The invalidate messages are for pages in the list associated with the lock being acquired that meet three criteria: 1) the page is cached at the client in the valid state, 2) the LSN of the page in the cpt for the client is smaller than the LSN of the page in the lock list, and 3) the client acquiring the lock was not the last to update the page under this lock. The invalidated pages are marked invalid in the cpt for the client and at the client site.

- **Page refresh:** When the server sends a page to a client (page refresh), at the server, the page is marked valid in the cpt for the client and the LSN for the page in the cpt is updated to be the LSN for the page at the server.

- **Lock list cleanup:** We are interested in keeping the list of pages with every lock as small as possible. This can be achieved by periodically deleting pages P from the list of lock L such that the following condition holds, where C is the client noted in the list of pages for L as the last client to update P:

  > Every client other than C has the page cached either in an invalid state or with LSN greater than or equal to the LSN for the page in the list for lock L.

  The rationale for this rule is that the purpose of region lock lists is to determine pages that must be invalidated. However, if for a page in a client's cpt, the LSN is greater than the LSN for the page in the lock list, then the client has the most recent update to the region on the page, and thus the page will not need to be part of any invalidation list sent to the client.

5 Shared Disk Recovery: Model and Common Structures

In the shared disk approach, a number of machines are interconnected and also have direct access to disks over a fast network. The shared disk environment is used in many systems, such as the DEC VAXclusters, and provides benefits over a shared nothing architecture, such as faster access to non-local disks and fault-tolerance. Also, the basic advantage of shared disk schemes over the client-server schemes is that the algorithms are symmetric with respect to which site executes them, preventing one system from becoming a bottleneck in the system. As in our client-server scheme, in addition to careful consideration of the interaction with multi-level recovery, our main concern is minimizing false sharing through fine-grained concurrency control. This allows, for example, read-only transactions with a fully cached working set to proceed at main-memory speeds, an important property for our intended applications.

We now describe our shared disk recovery model.

- Each site maintains its own copy of the entire database in memory and its own system log on disk. Thus, there may be multiple logs in the system.
- Sites obtain locks from a *Global Lock Manager* (GLM); the function of the lock manager could be distributed for speed and reliability, but this is orthogonal to our discussion.
- Sites cache locks, and relinquish locks based on the *call back* locking mechanism described in Section 4. We assume the network is FIFO and reliable.
- Each site has its own system log on disk and therefore the logs are distributed. To repeat history during restart recovery, we need some mechanism to temporally order log records that affect the same region. To enable this, each site maintains a global timestamp counter TS_ctr, and a timestamp obtained from this counter is stored in each physical redo log record for an update. We will see the details of how this TS_ctr is maintained and used later.
- Each site maintains its own version of the *dirty page table* dpt, system log (a stable portion on disk *and* a tail in memory), and an ATT which stores information relating to transactions that execute at that site (with separate undo and redo log records for each transaction).
- A single pair of checkpointed images is maintained on disk for the database. A checkpoint image consists of an image of the database, the dirty page table ckpt_dpt, and for every site:
  1. end_of_stable_log – the point in the site's system log from which the system log must be scanned during recovery.
  2. a copy of the ATT at the site (containing undo logs).

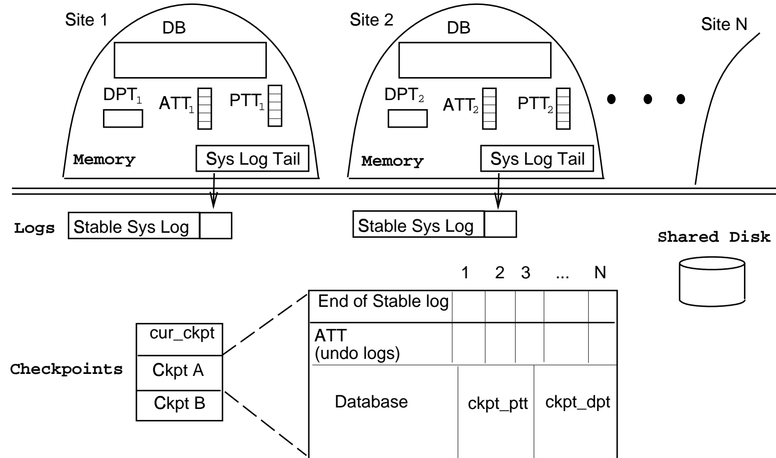

Figure 3: Page-Shipping Shared Disk Architecture

In the next two sections, we present two schemes for shared disk concurrency control and recovery. The first is a page-shipping approach which is similar in spirit to the Invalidate-on-Update client-server mode. The second is a log shipping scheme which allows concurrent use of non-overlapping regions on a page across sites.

6 Page-Shipping Shared Disk Recovery Scheme

Our page-shipping scheme is similar in spirit to the Invalidate-on-Update client-server scheme in that a transaction at a site updating a region on a page is guaranteed to have the latest copy of the page. Therefore, concurrent updates to different regions of a page are not possible in this scheme.

6.1 Data Structures

We now describe data structures specific to the page-shipping scheme. Common data structures were described in Section 5. An overview of the data structures for this scheme is given in Figure 3.

In addition to the TS_ctr for the site, a timestamp for each page is maintained at each site in the *page timestamp table*, ptt, which keeps track of the TS_ctr value when the page was last updated. Each page has an associated *page* lock which helps in ensuring that a transaction always has the latest copy of the page while accessing or updating the page. Sites cache locks, and relinquish locks based on the *call back* locking mechanism described earlier. Along with each of the two checkpoint images of the database is stored a checkpoint page timestamp table, referred to as ckpt_ptt.

6.2 Normal Processing

We describe below the actions taken during normal processing, in addition to those performed in the centralized case, to support distributed concurrency control and recovery. Checkpointing and recovery from system and site failure are described in subsequent subsections.

- **Update:** Like in the centralized case, before accessing a region, each transaction obtains a region lock from the LLM. Additional page locks are acquired in S(X) mode while accessing (updating) data on a page. If this lock is not cached at the site, actions are performed as described below under **Lock Acquisition**.

Page locks for an access are released by a transaction once the access is completed; page locks for an update are released by a transaction only after the update on the page is completed. The value of TS_ctr at the site when the redo log record was generated is stored in the redo log record corresponding to the update. Also, the timestamp for the updated page (in the ptt) at the site is set to the TS_ctr stored in the log record.

An important point to note is that log records in the system log may not be ordered on their TS_ctr values. This is because the value of TS_ctr is stored in the redo log record when the update is performed, but the log record is appended to the transaction local log, which is not flushed to the system redo log until operation or transaction commit.

- **Lock Release:** When a transaction releases an X mode region lock or operation lock, it stores the end of log in memory with the lock (this is stored to optimize the amount of flushing that needs to be done when a lock is relinquished, as in the client-server scheme). Note that all updates for the operation which held the region lock will be moved to the global log by the normal operation commit semantics *prior* to the release of this lock. Thus, for a region lock, all redo log records for updates to the region covered by the lock precede the end of log point stored with the lock (similar for operations). When a site relinquishes an X region lock or operation lock, it flushes the global log at its site until the end of log point stored with the lock. The flush on release of X region or operation locks is done to ensure that it is possible to repeat history during restart recovery, and appropriate locks for undoing operations are held in case of site crashes. Note that no flushes are performed when page locks are released.

  Additionally, when a site releases an X page or X region lock back to the GLM, it stamps it with the site's TS_ctr; the TS_ctr value of the lock is used by other sites that later acquire the lock, as we will see shortly. The GLM also stores with each page lock the site that last held the page lock in X mode; the information is updated each time a site relinquishes an X mode page lock,

- **Lock Acquisition:**

  A transaction acquiring a lock cached by the LLM need take no special action. If it is a page lock, then the page is already current at this site.

  When an X-mode page or region lock arrives from the GLM, it includes the timestamp from the last site that held the lock in X mode, as described above. Upon receiving an X region lock or page lock at a site, the site's TS_ctr is set to the maximum of 1) it's current value, and 2) the TS_ctr value associated with the incoming lock plus one.

  When a site acquires a page lock on behalf of a transaction from the GLM (that is, the lock is not already cached at the site), the site requests the page from the last site that held the page lock in X mode (using the site identifier sent with the lock). In order to handle single-site recovery, failure of the acquiring site to obtain a copy of the page, due to a failure of the site from which it is being requested, causes the lock acquisition to fail and the lock to be returned to the GLM unchanged.

Shipping timestamps with page locks ensures that log records for successive updates to a page at different sites are assigned increasing timestamp values. Shipping timestamps with region locks ensures that log records generated under conflicting locks are applied in the correct order during recovery even though redo log records in the individual site may not be ordered by timestamp (as mentioned earlier). However, the algorithm still works correctly, as shown in the discussion of recovery and correctness below.

6.3 Checkpointing

Unlike the centralized and client-server scheme, checkpointing in the shared disk environment requires coordination among the various sites. As mentioned above, a single pair of checkpointed images is maintained for all the sites.

The site initiating the checkpoint coordinates the operation, which consists of the following three steps at each site – 1) writing the database file image 2) writing the ATT and 3) flushing the global log. Below, we describe each step:

1. The coordinator announces the beginning of the checkpoint, at which time all sites (including the coordinator) note their current end_of_stable_log values, then make a copy of their dpts and zero their dpts. Note that zeroing the dpt and recording end_of_stable_log is done atomically with respect to flushes.

   Each site then makes a copy of its current ptt and sends it to the coordinator along with the end_of_stable_log (noted above), and a copy of the dpt. The coordinator constructs ckpt_dpt by or'ing together the copy of its dpt and all the dpts received from other sites (recall that we are assuming the dpt is a bitmap). The database pages to be written out during the checkpoint are the pages that are dirty in ckpt_dpt or in the ckpt_dpt in the previous checkpoint.

   For each page to be written out, the coordinator uses the ptts sent to it by the other sites and its own ptt to determine the site whose ptt contains the highest timestamp for the page. This site is responsible for writing the page to the checkpoint image. Once the coordinator has partitioned the set of pages to be written out among the various sites, each site is sent the set of page identifiers assigned to it. A site, upon receiving its assigned set of pages to write, proceeds to write those pages to the checkpoint image. Since no two sites will be assigned the same page, site can write pages concurrently.

   The coordinator then constructs ckpt_ptt by first reading the ckpt_ptt in the previous checkpoint into memory. For every page that was determined to be written out (by some site i), the timestamp for the page in ckpt_ptt is set to its timestamp in the copy of the ptt for site i. Finally, ckpt_dpt constructed earlier, ckpt_ptt and the end_of_stable_logs for all the sites are written to the checkpoint.

   Note that since the site with the highest timestamp for a page writes the page to the checkpoint image, updates to the page by log records preceding end_of_stable_log recorded for a site, are contained in the checkpoint. Furthermore, as will be discussed in the correctness section below, updates for a page recorded in log records with timestamps less than the timestamp for the page in ckpt_ptt are also contained in the checkpoint.

2. Once every site has written out the database image and reported this to the coordinator, the coordinator instructs each site to write out its ATT. Note that multiple sites can be concurrently writing out the ATT.

3. After writing out the ATT, each site flushes the global log at that site as in the centralized case. Finally, the database checkpoint is committed after all sites have completed their flushing.

6.4 Recovery

In case the entire system fails, *restart recovery* is performed by any one site, say j. The site j, which we will call the *acting coordinator* site, reads the following from the most recent checkpoint image: the database image, the ckpt_ptt, and for each site, the ATT and the end_of_stable_log. A separate page table ptt is initialized to ckpt_ptt and for each site i a separate dpt, dpt_i is initialized to contain zero bits for all pages. Starting from the end_of_stable_log point stored for a site in the checkpoint, the log records in all

the system logs are merged as described below, and applied to the database. To merge the system logs, they are scanned in parallel; at each point, if the next log record in any of the system logs is not a redo log record, then any one such record is processed and the ATT for its site is modified as described for the centralized case in Section 2.7. On the other hand, if the next records in all the system logs are redo log records, then the log record output next is the one amongst them with the lowest timestamp value. If, for a page updated by the log record, the timestamp in the log record is greater than or equal to the timestamp for the page in ckpt_ptt, then 1) the update is applied to the page, 2) the page is marked dirty in the dpt for the site whose system log contains the record, and 3) the timestamp for the page in ptt is set to the maximum of its current value and the timestamp in the log record.

Note that redo records in the system log for a site may not be in timestamp order as mentioned earlier. However, this does not cause a problem and conflicting log records are applied in the order in which they were generated. The reason for this is that for two conflicting log records in separate system logs, the earlier log record and log records preceding it in its system log have lower timestamps than the log record generated later. This fact is revisited below in our overview of correctness.

Once the last log record has been processed, TS_ctr at the acting coordinator site j is set to the largest timestamp contained in the ptt at site j. Site j then rolls back in-progress operations in the ATTs for the various sites beginning with level L_0 and then considering successive levels L_1, L_2 and so on (as described in Section 2.7). When an operation in an ATT entry for a site i is being processed, actions are performed on the undo and redo logs for the entry. Furthermore, each redo log record generated when processing an operation for site i is assigned a timestamp equal to TS_ctr at site j, and when an operation pre-commits/aborts, log records from the redo log are appended to the system log for site i.

Next, site j flushes every site's system logs causing appropriate pages in the dpt for the site (maintained at site j) to be marked dirty. After this point, the other sites are involved in recovery. The TS_ctr at every site is set to the TS_ctr at site j after incrementing it by one. The dpt at each site is then set to the dpt maintained for the site during recovery at site j, and the database image and ptt at each site is set equal to the database image and ptt at site j. Finally ckpt_ptt and dpt for other sites are deleted from site j, bringing recovery to completion.

6.5 Overview of Correctness

In this section, we present additional arguments about the correctness of our page-shipping recovery scheme by discussing below several properties on which the correctness is based.

1. A page, i, in a checkpoint image reflects all updates with timestamp less than ckpt_ptt[i].
2. Any log record affecting page i prior to end_of_stable_log at any site has timestamp less than or equal to ckpt_ptt[i] and is reflected in the checkpoint image of page i.
3. If L_1 and L_2 are conflicting log records and L_1 is generated before L_2, then if L_2 is flushed to the stable log, then so is L_1.
4. If L_1 and L_2 are conflicting log records in different system logs and L_1 is generated before L_2, then L_1 and all log records preceding it in its system log have lower timestamps than L_2.

(1) follows from the fact that timestamps for pages in the ptt are set only after they are updated, and passing timestamps with page locks guarantees that successive updates to a page have non-decreasing timestamps (and in turn, assign non-decreasing timestamps to the ptt entry).

(2) For a log record that updates page i prior to end_of_stable_log at a site, (a) ppt[i] at the site is greater than or equal to the timestamp in the log record, (b) the page is in the dpt of the site and (c) the page at the site contains the update (when the site sends its ppt and dpt to the coordinator during the first phase of the checkpoint). Thus, since the site for which ppt[i] is the largest writes the page to

the checkpoint image, ckpt_ppt[i] is greater than or equal to the timestamp in the log record. Also, since versions of a page with higher timestamps contain all updates in versions with lower timestamps, the update by the log record is reflected in the checkpoint image of page i.

(3) When the region lock covering L_1 was released, it must have followed the commit of an operation due to the rules of multi-level recovery. Thus, that log record would be moved to the global log during the operation commit, and would thus be before the point noted on release of the region lock. The flush to that point carried out when a region lock is released by a site guarantees the property.

(4) The shipping of TS_ctrs with region locks ensures this property. The reason for this is that L_1 is appended to the system log at its site before the X region lock for the updated region is released by the site. The timestamp assigned to the lock is the TS_ctr at the site for L_1 which is at least as large as the timestamps in L_1 and all the redo records preceding L_1 in the releasing site's system log. Before L_2 can be generated, its site has to acquire the region lock in X mode which causes the timestamp at the site where L_2 is generated to be set higher than the timestamp for the lock. Thus, L_2 is assigned a timestamp greater than L_1 as well as all log records preceding L_1.

Properties (1) through (4) can be used to show that our recovery scheme repeats history when scanning the system logs. Property (1) implies that updates to a page i by a log record do not need to be applied if ckpt_ptt[i] is greater than the timestamp in the log record. From property (2), it follows that log records preceding end_of_stable_log can be ignored since these updates are already contained in the checkpoint image. Property (3) ensures that log records in the system logs accurately and completely describe the history of updates to every region. Finally, property (4) ensures that conflicting updates described by log records that appear after end_of_stable_log are applied during recovery in the order in which they were performed during normal processing (in spite of timestamps possibly being out of order within a single site's log). Note that, for conflicting log records L_1 and L_2 on page i, L_1 generated before L_2, L_2 may precede end_of_stable_log for its site, while L_1 follows end_of_stable_log for its site. In this case, due to property (2), ckpt_ptt[i] would be greater than or equal to the timestamp for L_2 and the timestamp for L_2 would be greater than that for L_1. Thus, the update to page i by L_1 would not be applied during recovery.

6.6 Recovery from Site Failure

Our recovery algorithm can also be extended to deal with a site failure *without* performing a complete system restart, so long as the GLM data has not been lost, or can be regenerated from the other sites. If this is not the case, a full system recovery is performed instead.

Recovery from a single site failure is complicated since log records for updates by active transactions at the failed site may not have made it to stable storage while the updates themselves may have been propagated to other sites when the pages containing the updates are shipped between sites. There is no way to undo these updates since the undo information for them is contained is main-memory and is lost when the site failed. Thus, the only way to roll back the above set of updates is to recover the set of pages that the updates span. (Note that such a problem would not arise with a scheme providing lower concurrency, such as page locks held to end of transaction.)

In order to support this roll-back, it must be possible to associate with any region or operation lock the set of pages such that some part of the page may be updated by an operation that holds the lock; we call this set of pages as the pages *affected* by the lock.

The first step, when a site j recovers, is to determine the set of pages that must be recovered—these are pages that either:

1. May contain updates by uncommitted transactions from site j, or
2. Were last updated by site j.

The pages in (1) are those affected by any operation or X mode region lock held by site j at the GLM. The pages in (2) are those pages on which site j was the last site to have obtained an X page lock. The pages in (2) that are not in (1) are the set of pages that contain updates belonging to transactions that committed at site j. Note that for these pages, if a different site k holds an S lock on the page, the page need not be recovered, and the GLM merely notes that the site k has the latest version of the page.

Once the set of pages to be recovered are determined as described above, they are all locked in X mode by site j so that all updates to these pages by other sites are blocked (any other page locks held by site j are released).

Site j then retrieves from the most recent checkpoint, the database image, the ATT for site j, ckpt_ptt and the end_of_stable_log for each site. It then requests from every other site, the current end_of_stable_log at the site and the sequence of redo records in memory at the site (that is, redo records in the transaction local logs or in the system log after end_of_stable_log) involving updates to the pages being recovered. The redo pass is performed by scanning all the system logs as described in the Section 6.4 except that 1) only updates to pages being recovered are applied depending on the timestamps for these pages in ckpt_ptt, and the timestamps for only these updated pages are modified in ptt, 2) only the pages in the dpt for site j are marked dirty, 3) only actions on the ATT for site j are performed, and 4) the system log for a site is scanned until the end_of_stable_log returned by that site at the beginning of this recovery. After this, the in-memory redo records received from the various sites are applied in timestamp order to the pages being recovered, and the timestamps for the updated pages in ptt are set to the timestamp in the log record.

At the end of the redo pass, the pages being recovered contain updates by transactions at every other site and updates by transactions at site j for which log records are contained in the stable log (thus, updates described by redo log records in memory of site j when it crashed are absent – this includes the transaction local logs and the portion of the global log in main memory). At this point, other sites can be granted page locks held by site j if they request it. TS_ctr at site j is set to be greater than the largest timestamp in the ptt at site j.

Before rolling back in-progress operations, the locks that were cached at site j at the time it crashed are obtained by the recovery process at site j by consulting the GLM. As described in Section 2.7, rollback is performed level by level, with additional locks requested as is done during normal processing (see Section 7.2). Level L_i operation locks at site j can be released once all active operations at level L_{i+1} have been rolled back.

As in normal processing, TS_ctr at site j is incremented when a new lock is obtained, TS_ctr stored in a redo log record and in the timestamp entries for updated pages when the redo log is generated, and log flushes are performed when operation/X mode region locks are released by site j.

7 Log-Shipping Shared Disk Recovery Scheme

We are interested in improving the concurrency of the page-shipping shared disk recovery scheme by allowing multiple concurrent readers and writers of the same page at different sites, as long as the parts of the page they update come under different region locks. A result of this is that copies of a page at different sites may contain a different set of updates, which must be merged before the page is written to disk. Unlike the client-server case, there is no server to carry out the task of merging updates.

To solve the above problem, in our scheme, log records generated at a site are broadcast to all other sites, so the updates can be carried out there. Since log records are shipped, there is no need to ship pages. The scheme ensures that every time a site obtains a region lock, the most recent version of the region is guaranteed to be accessed at the site. More precisely, it guarantees that every time a site obtains any lock (whether an operation lock or a region lock), all log records generated by all operations which held the same lock in a conflicting mode have been applied to the local page images.

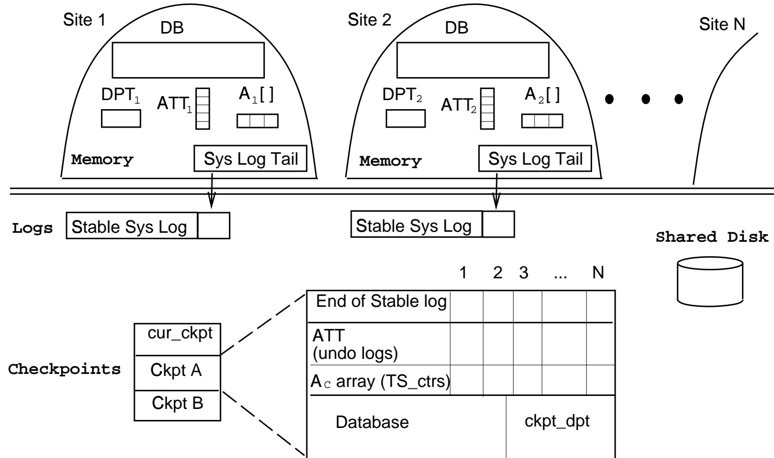

Figure 4: Log-Ship Shared Disk Architecture

The idea of broadcasting log records leads to an architecture that essentially implements distributed shared memory, without the overhead of shipping pages. Note that the overhead of broadcasting log records to all the sites may not be too severe if update rates are not too high. Broadcasting may also be seen as a strategy to propagate updates early, possibly using greater bandwidth, but avoiding the latency of waiting for pages to be shipped when another transaction needs to update the data. Also, in some network architectures (e.g., ethernet), the cost of a broadcast to all sites may not be very different from the cost of sending a message to a single site.

7.1 Data Structures

An overview of data structures used for our shared disk scheme is given in Figure 4. In addition to the common elements described in Section 5, the log-based scheme maintains the following additional data structures. At every site j, an array of TS_ctrs (one TS_ctr per site), A_j is maintained in memory. $A_j[i]$ stores the timestamp of the latest update from site i that has been applied to the database at site j. A_j serves a purpose similar to ptts in the page-shipping scheme – it keeps track of the state of the database relative to log records.

With each checkpointed image on disk, each site stores the TS_ctr following which redo log records from that site must be applied to the database. Collectively these counters are referred to as A_C. Note that since pages are not shipped between sites, the log-shipping scheme does not need page locks.

7.2 Normal Processing

We describe below the actions taken during normal processing to support distributed concurrency control and recovery (in addition to those for the centralized case). Checkpointing and recovery from system and site failure are described in subsequent sections.

- **Log Records:** Every time a physical redo log record is moved from a transaction's local redo log to the system log, TS_ctr is incremented by 1 and stored in the log record. The timestamps are used to order log records that describe conflicting updates.
- **System Log Flush:** When the system log at site i is flushed to stable storage, each redo log record which has hit the disk is also broadcast to the other sites. The sending site i, also sets $A_i[i]$ to the timestamp in the log record. Flushing of a sequence of log records is completed once

every log record has been written to disk as well as sent to the remaining sites. Also, as in the centralized case, pages updated by the flushed log records are marked dirty in the site's dpt.

- **Log Record Receipt:** A site j processes an update broadcast to it from site i as follows (updates are processed in the order in which they are received). On receiving a broadcast log record, the site applies the update to its local copy of the affected page(s), and sets the appropriate bits in its dpt. After updating the appropriate pages, the site sets $A_j[i]$ to the timestamp contained in the update (redo log record).

- **Lock Release:** The lock managers aid the scheme in two ways. First, as in the previous schemes, the current local end-of-log is noted with region and operation locks when the lock is released by a transaction, and the LLM ensures that the log is flushed to this point before releasing the lock from the site. This aids in recovery by ensuring that history is repeated, and when lower level locks are released, the logical undo actions which accompany the higher level locks have made it to disk. Since logs are broadcast on flush, it helps ensure that another site will receive the necessary log records before getting the same lock in a conflicting mode.

  Note that the FIFO property of the network does not ensure that a site j receives an update broadcast from a site i before it obtains the region lock for the updated region from the GLM (relinquished to the GLM by site i). In order to ensure that all previous updates to a region are received by a site before it obtains the region lock, the LLM before releasing a region lock from a site must ensure that not only have all redo log records preceding the end-of-log (noted for the lock) flushed to disk, but also that acknowledgments of receipt of the broadcast records have been received from all sites.

  Second, when a transaction releases an X mode region lock, the timestamp for the lock is set to the current value of TS_ctr at the site. When this lock is called back by the GLM, this value is also sent and is associated with the lock by the GLM. When received by another site, the timestamp is used to ensure that log records for conflicting actions covered by this lock have increasing timestamp values. As an optimization, the site identifier can also be sent with the lock to the GLM; the purpose will become clear in the next point.

- **Lock Acquisition:** When a site receives an X mode region lock from the GLM, it sets its own TS_ctr to be the maximum of its current TS_ctr and the timestamp associated with the lock (received from the GLM). Further, the lock is granted to a local transaction only after all outstanding (unapplied) updates at the time of acquiring the lock have been applied to the page. This is to ensure that data accessed at a site is always the most recent version of the data.

  As an optimization, if a site identifier is provided with the lock by the GLM, it suffices to process log records up to (and including) the log record from the site with the timestamp provided.

7.3 Checkpointing

Checkpointing is coordinated by one of the sites. The checkpointing operation consists of three steps — 1) writing the database image by the coordinator, 2) writing the ATT at each site and 3) flushing the logs at each site. The main difference from the centralized case lies in how each step is carried out. We describe each step below:

1. The coordinator announces the beginning of the checkpoint, at which time all other sites first make a copy of their dpts and then subsequently zero their dpts, and note their current end_of_stable_log values. Note that recording end_of_stable_log and dpt, and then zeroing dpt is done atomically with respect to flushes. Every site sends the recorded dpt and end_of_stable_log values to the coordinator.

The coordinator site j applies all outstanding updates, then atomically (with respect to processing further log records and flushing) records its end_of_stable_log, copies its timestamp array A_j to A_C, copies its dpt to ckpt_dpt, and then zeroes its own dpt. The coordinator then or's its ckpt_dpt with the copies of the dpts it receives from the other sites. It then writes to the checkpoint image the ckpt_dpt, the end_of_stable_logs for each site, and the timestamp array A_C.

Next, the database image is written out by the coordinator in the same fashion as in the centralized case, writing out not only pages dirty in this checkpoint interval (in ckpt_dpt), but also pages dirtied in the previous checkpoint interval (in the ckpt_dpt stored in the previous checkpoint).

2. Once the coordinator has written out the database image, it instructs each site to write out its ATT. Multiple sites can be concurrently writing out their ATTs.

3. The logs are flushed at each site and after all sites flush their logs the coordinator commits the checkpoint by toggling cur_ckpt, as in the centralized case.

Note that in Step 1, applying outstanding updates at the coordinator before recording ckpt_dpt and A_C ensures that updates preceding end_of_stable_log reported by other sites have been applied to the database pages, and thus, it is safe to zero dpts at sites when end_of_stable_log is noted. Also, since each site notes end_of_stable_log independently, it is possible that for a redo log record after end_of_stable_log at one site, a conflicting redo log record generated after it may be before end_of_stable_log noted at a different site. As a result, during restart recovery, applying every update after end_of_stable_log in the system log for a site could result in the latter update being lost. Storing A_C in the checkpoint and during restart recovery, applying only redo records at site i whose timestamps are greater than $A_C[i]$ eliminates the above problem since timestamps for both updates would be smaller than the corresponding TS_ctr values for the sites in A_C.

7.4 Recovery

Restart recovery in case of a system wide failure (where all sites have to be recovered) can be performed as follows by an arbitrary site j in the system, which we will call the acting coordinator. The following actions are performed by site j alone.

First, the database image and the checkpointed timestamp array A_C are read, and for each site, the ATT and the end_of_stable_log recorded in the checkpoint are read. Redo log records in the system logs for the various sites are then applied to the database image by concurrently scanning the various system logs. Each site's system log is scanned in parallel, starting from the end_of_stable_log recorded for the site in the checkpoint. At each point, if the next log record to be considered in any of the system logs is not a redo log record, then it is processed and the ATT for its site is modified as described for the centralized case in Section 2.7. On the other hand, if the next record to be considered in all the system logs is a redo log record, then the log record considered next is the one (among all the system logs on disk being considered) with the lowest timestamp value. For every redo log record encountered in the system log for a site, i, with a timestamp greater than $A_C[i]$, the update is applied and the affected pages are marked as dirty in j's dpt.

Once all the system logs have been scanned, TS_ctr at site j is set to the largest timestamp contained in a redo log record. In-progress operations in the ATTs for the various sites are then rolled back and executed, respectively, at site j against the database at site j, beginning with level L_0 and then considering successive levels L_1, L_2 and so on (as described in Section 2.7). When an operation in an ATT entry for a site is being processed, actions are performed on the undo and redo logs for the entry. Furthermore, when an operation pre-commits/aborts, log records from the redo log are appended to the system log for the site and the timestamp for each redo log record appended is obtained by incrementing TS_ctr at site j.

Finally, every site's system logs are flushed causing appropriate pages in j's dpt to be marked dirty (updates are not broadcast, however), and the TS_ctr at every site and $A_k[i]$ for all sites k and i are set to the TS_ctr value at site j. The database image at every site is copied from the database image at site j, the dpt for each site is copied from the dpt at site j; recovery is then complete.

7.5 Overview of Correctness

The correctness of the checkpointing and recovery algorithms follows from the following properties.

1. If the timestamp contained in a log record for site i is less than or equal to $A_C[i]$, then the log record's effects must have made it to the copy of the database in the checkpoint.
2. Any log record in the system log for site i prior to end_of_stable_log for the site has a timestamp less than or equal to $A_C[i]$.
3. If L_1 and L_2 are conflicting log records and L_1 is generated before L_2, then if L_2 is flushed to the stable log, then so is L_1.
4. If L_1 and L_2 are conflicting log records in different system logs and L_1 is generated before L_2, then L_1 has a lower timestamp than L_2.

Property (1) holds since when a page is written to disk during a checkpoint at site j, updates preceding $A_j[i]$ have made it to the image of the page at site j (due to the algorithm for application of incoming log records), and this page is dirty in j's dpt (because the dpt is noted atomically with A_C).

Property (2) holds since before site i sends its end_of_stable_log to the checkpoint coordinator, any update preceding it is sent to the coordinator (when the system log is flushed at site i). Since the network is FIFO, the receipt of the end_of_stable_log implies that all necessary log records have arrived. Furthermore, before noting $A_C[i]$, the coordinator applies outstanding updates from site i and thus sets $A_j[i]$ to the timestamp of the last update applied from site i.

Property (3) holds since the log is flushed every time a site relinquishes a region lock. Finally, property (4) holds since before the region lock that guards L_1 is released by its site, L_1 is appended to the system log and assigned a timestamp from the TS_ctr at the site. Furthermore, the TS_ctr at L_1's site is shipped along with the region lock when it releases the region lock, and the site for L_2 sets its TS_ctr to be at least the timestamp it receives when it acquires the lock. Thus, since L_2 is generated after the lock is obtained by its site, it is assigned a timestamp greater than the TS_ctr value at its site when the site receives the region lock, and the property holds.

From the above properties, it follows that history is repeated as a consequence of applying the redo log records contained in the system logs in timestamp order during restart recovery. From properties (1) and (2), it follows that log records preceding end_of_stable_log can be ignored since these updates are already contained in the checkpoint image. Similarly, property (2) implies that updates at a site i by a log record do not need to be applied if $A_C[i]$ is greater than or equal to the timestamp in the log record. Property (3) ensures that log records in the system logs accurately and completely describe the history of updates to every region. Finally, property (4) ensures that conflicting updates described by log records that appear after end_of_stable_log are applied during recovery in the order in which they were performed during normal processing. Note that, for conflicting log records L_1 and L_2, L_1 generated before L_2, L_2 may precede end_of_stable_log for its site (say i), while L_1 follows end_of_stable_log for its site. In this case, due to property (2), $A_C[i]$ would be greater than or equal to the timestamp for L_2 and the timestamp for L_2 would be greater than that for L_1. Thus, the update by L_1 would not be applied during recovery.

7.6 Recovery from Site Failure

Our recovery algorithm can also be extended to deal with a site failure *without* performing a complete system restart, so long as the GLM data has not been lost, or can be regenerated from the other sites. If this is not the case, a full system recovery is performed instead. Recovery from site failure, as with regular system recovery, has a redo pass, followed by rollback of in-progress operations.

Before beginning the redo recovery pass, the recovering site, say j, retrieves from the most recent checkpoint the database image, the ATT for site j, the timestamp array A_C and the end_of_stable_log for each site. It then informs other sites that it is up, and requests from each site i, that site's current end_of_stable_log value, and the value of $A_i[j]$. At this point, other sites start sending log records to j; these are buffered and processed later. The redo pass is then performed by scanning all the system logs as described in the previous subsection except that 1) only the pages in the dpt for site j are marked dirty, 2) only actions on the ATT for site j are performed, and 3) the system log for a site is scanned until the end_of_stable_log returned by that site at the beginning of site j's recovery.

Also, log records in the tail end of the log of the recovering site may not have made it to other sites – since a log record is broadcast after it is flushed. For each site i (other than the recovering site, j) all log records in site j's system log that have timestamps greater than $A_i[j]$ are broadcast to site i as they are processed. Once the redo pass is completed, $A_j[i]$ is set to the maximum timestamp in a redo log record encountered during the redo pass in the system log for site i. Also, TS_ctr at site j is set to the maximum of $A_j[i]$ for all sites i. At this point, site j can begin applying updates described by log records received from other sites, as during normal processing, in the order received, and checkpoints can again be taken as normal.

Before rolling back in-progress operations, the locks that were cached at site j at the time it crashed are re-obtained by the recovery process at site j by consulting the GLM. As described in Section 2.7, rollback is performed level by level, with additional locks requested as is done during normal processing (see Section 7.2). Thus, TS_ctr at site j is incremented and outstanding updates are applied when a new lock is obtained, TS_ctr is incremented when a redo log record is appended to the system log, and log flushes are performed when operation/X mode region locks are released by site j. Also, level L_i operation locks at site j can be released once all active operations at level L_{i+1} have been rolled back.

8 Conclusion

In this paper, we showed how our single-site multi-level recovery algorithm for main-memory databases can be extended to a distributed-memory data-shipping system while maintaining many of the original benefits of the single-site algorithm. The first scheme presented supports client-server processing in which a central system controls logs and checkpoints. In the second and third scheme, suitable for a cluster of computers with a shared disk, sites participate symmetrically in transaction processing activities.

We described details of recovery after the failure of clients or the server in the client-server case, and from single site as well as system-wide failure in the shared disk case. Our schemes allow concurrent updates at multiple clients in a client-server environment or multiple sites of the shared disk environment. By allowing fine-grained and flexible concurrency control, our schemes are applicable to a range of distributed, main-memory applications which need transactional access to data.

Our distributed schemes are based on a multi-level scheme for recovery in main-memory databases which has been implemented in the Dalí Main Memory Storage Manager [9]. Thus, the benefits of this algorithm are extended to the distributed schemes; the benefits include fuzzy checkpointing, use of the log for implementing functions that otherwise require page latching, low overhead logging with undo records written only due to a checkpoint, and per-transaction logs for low contention.

Future work includes parallelization of recovery in the shared disk setting, and recovery in a system

where not all sites store the entire database. We also plan to explore the performance of our schemes through experimentation, and then build a distributed, data-shipping version of Dalí based on these algorithms.

References

[1] P. Bohannon, D. Lieuwen, R. Rastogi, S. Seshadri, S. Sudarshan, and A. Silberschatz. The architecture of the Dali main-memory storage manager. *Multimedia Tools and Applications*, 4(2):115–151, March 1997.

[2] P. Bohannon, J. Parker, R. Rastogi, S. Seshadri, and S. Sudarshan. Distributed multi-level recovery in main-memory databases. Technical Report 112530-96-02-27-01TM, Lucent Technologies, Bell Laboratories, February 1996.

[3] M. J. Carey, D. J. DeWitt, M. J. Franklin, N. E. Hall, M. L. McAuliffe, J. F. Naughton, D. T. Schuh, M. H. Solomon, C. K. Tan, O. G. Tsatalos, S. J. White, and M. J. Zwilling. Shoring up persistent applications. In *Proceedings of ACM-SIGMOD 1994 International Conference on Management of Data, Minneapolis, Minnesota*, pages 383–394, May 1994.

[4] M. J. Carey, M. J. Franklin, and M. Zaharioudakis. Fine-grained sharing in a page server OODBMS. In *Proceedings of ACM-SIGMOD 1994 International Conference on Management of Data, Minneapolis, Minnesota*, pages 359–370, May 1994.

[5] D. J. DeWitt, R. Katz, F. Olken, D. Shapiro, M. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. *Proc. ACM-SIGMOD 1984 Int'l Conf. on Management of Data*, pages 1–8, June 1984.

[6] M. J. Franklin, M. J. Zwilling, C. K. Tan, M. J. Carey, and D. J. DeWitt. Crash recovery in client-server EXODUS. In *Proceedings of ACM-SIGMOD 1992 International Conference on Management of Data, San Diego, California*, pages 165–174, June 1992.

[7] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):509–516, December 1992.

[8] Robert B. Hagmann. A crash recovery scheme for a memory-resident database system. *IEEE Transactions on Computers*, C-35(9):839–847, September 1986.

[9] H.V. Jagadish, Dan Lieuwen, Rajeev Rastogi, Avi Silberschatz, and S. Sudarshan. Dali: A high performance main-memory storage manager. In *Procs. of the International Conf. on Very Large Databases*, 1994.

[10] H.V. Jagadish, Avi Silberschatz, and S. Sudarshan. Recovering from main-memory lapses. In *Procs. of the International Conf. on Very Large Databases*, 1993.

[11] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The objectstore database system. *Communications of the ACM*, 34(10), October 1991.

[12] T. Lehman, E. J. Shekita, and L. Cabrera. An evaluation of Starburst's memory resident storage component. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):555–566, December 1992.

[13] D. Lomet. MLR: A recovery method for multi-level systems. In *Proceedings of ACM-SIGMOD 1992 International Conference on Management of Data, San Diego, California*, pages 185–194, 1992.

[14] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1):94–162, March 1992.

[15] C. Mohan and I. Narang. Recovery and coherency-control protocols for fast intersystem page transfer and fine-granularity locking in a shared disks transaction environment. In *Proceedings of the Seventeenth International Conference on Very Large Databases, Barcelona*, pages 193–207, September 1991.

[16] C. Mohan and I. Narang. ARIES/CSA: a method for database recovery in client-server architectures. In *Proceedings of ACM-SIGMOD 1994 International Conference on Management of Data, Minneapolis, Minnesota*, pages 55–66, May 1994.

[17] E. Rahm. Recovery concepts for data sharing systems. In *Proceedings of the Twenty first International Conference on Fault-Tolerant Computing (FTCS-21), Montreal*, pages 109–123, June 1991.

[18] K. Salem and H. Garcia-Molina. System M: A transaction processing testbed for memory resident data. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):161–172, March 1990.

[19] K. Salem and H. Garcia-Molina. System M: A transaction processing testbed for memory resident data. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):161–172, 1990.

[20] G. Weikum, C. Hasse, P. Broessler, and P. Muth. Multi-level recovery. In *Proceedings of the Nineth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Nashville*, pages 109–123, June 1990.