Distributed Multi-Level Recovery in Main-Memory Databases

Philip Bohannon[*]
James Parker[*]
Rajeev Rastogi[*]
S. Seshadri[†]
Avi Silberschatz[*]
S. Sudarshan[†]
∗ Bell Laboratories, Murray Hill, NJ
{plbohannon,rastogi,avi}@bell-labs.com
parker@lucent.com
† Indian Institute of Technology, Bombay, India
{seshadri,sudarsha}@cse.iitb.ernet.in

Abstract

In this paper, we present two schemes for concurrency control and recovery in distributed *main-memory databases. In the client-server scheme, clients ship log records to the server, which applies the updates to its database copy. In the shared disk scheme, each site broadcasts its updates to other sites. The above enable our schemes to support concurrent updates to the same page at different sites.*

Both schemes support an explicit multi-level recovery abstraction for high concurrency, reduced disk I/O by writing only redo log records to disk during normal processing, and use of per-transaction redo and undo logs to reduce contention. Further, we use a fuzzy checkpointing scheme that writes only dirty pages to disk, yet minimally interferes with normal processing, not requiring updaters to even acquire a latch before updating a page.

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., [GMS92, LSC92, JLR$^+$94, DKO$^+$84]), 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.

Further performance improvements can be obtained for a number of applications by employing a distributed architecture in which several machines connected by a fast network perform database accesses and updates in parallel. This is especially the case in applications in which transactions are predominant-

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

ly read-only and update rates are low (e.g., number translation and call routing in telecommunications). As a result, each machine can locally access data cached in memory, thus avoiding network communication which could be fairly expensive. A very different example is CAD processing, in which locality of reference is very high, update transactions are long, and interactive response time is very important. Finally, distribution also enhances fault tolerance, which is required in many mission-critical applications even if data fits easily in main-memory. In this case, especially with low update rates, a distributed database is preferable to a hot-spare since load can be distributed in the non-failure case leading to improved performance.

The goal of the work described here was to extend the main-memory recovery scheme presented in [JSS93, BPR+96] to the distributed case, maintaining the efficiencies of the single-site scheme, and supporting the applications described above. For example, we can make use of the MMDB optimization called *transient undo logging*, originally proposed in [JSS93], in which undo log records are kept in memory and only written to disk as required for checkpointing. This reduces the size of the log written to disk, and perhaps more importantly, the size of the log sent across network links in distributed protocols.

We present two distinct but related distributed recovery schemes, the first for *client-server* architectures and the second for *shared disk* architectures. These are both "data-shipping" schemes (e.g., [FZT+92]) 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 our schemes is that concurrent updates are possible at granularities smaller than a page-size, minimizing false-sharing (and thus needless network accesses). In addition to the *transient redo logging* optimization described above, our algorithms provide advanced features such as explicit multi-level recovery (e.g., [WHBM90, MN94, Lom92]), and *fuzzy checkpointing* [SGM90a, Hag86]. Site or global recovery requires only a single pass over the system log, starting from the end of the system log recorded during the most recent checkpoint. As mentioned earlier, objects in the system can span one or more page boundaries.

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. We present our client-server recovery algorithm in Section 3, and the shared disk algorithm in Section 4. Related work and our conclusions are presented in Sections 5 and 6, respectively.

2 Overview of Main-Memory Recovery

In this section we present a review of multi-level recovery concepts and an overview of our single-site main-memory recovery scheme. Our centralized scheme extends the scheme presented in [JSS93] with multi-level recovery, and a fuzzy checkpointing scheme that only writes dirty pages. Low-level details of our scheme are described in [BPR+96].

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 with exclusive (X) and shared (S) modes, referred to as the *region lock*, that guards accesses and updates to the region.

2.1 Multi-Level Recovery

Multi-level recovery [WHBM90, MHL+92, Lom92] 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 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 specific updates at specific positions, this typically implies that the region locks on the updated index nodes be held 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 are retained. For example, region locks on the particular nodes involved in an insert can be

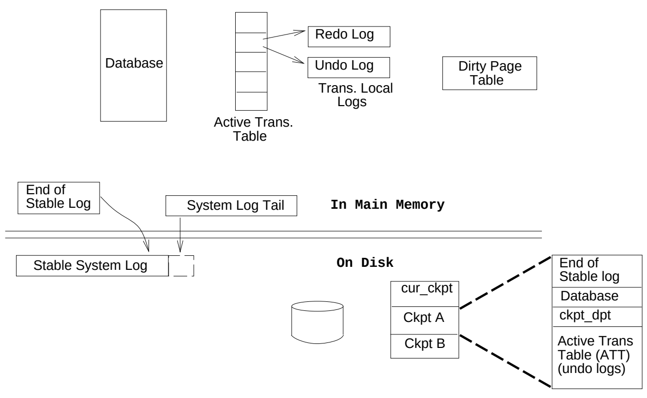

Figure 1: Overview of Recovery Structures

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 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) that stores separate redo and undo logs for active transactions. A dirty page table, dpt, is maintained in memory which records the pages that have been updated since the last checkpoint. The ATT (with undo logs) and the dirty page table are also stored with each checkpoint. The dirty page table in a checkpoint is referred to as ckpt_dpt.

2.3 Transactions and Operations

Transactions, in our model, consist of a sequence of operations. Similar to [Lom92], we assume that 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. We use the same terminology for operations, where only the pre-commit point is meaningful, though this is sometimes referred to as "operation commit" in the paper.

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

2.4 Logging Model

The recovery algorithm maintains separate 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 update (to a part of a region) generates physical undo and redo log records that are appended to the transaction's undo and redo logs respectively. When a transaction/operation pre-commits, all the redo log records for the transaction in its redo log are appended to the system log, and the logical undo description for the operation is included in the operation commit log record in 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 deleted from the transaction's undo log and a logical undo log record containing the undo description for the operation is appended. In-memory undo logs of transactions that have pre-committed are deleted since they are not required again. Locks acquired by an operation/transaction are released once they pre-commit.

The system log is flushed to disk when a transaction decides to commit. Pages updated by every redo log record written to disk are marked dirty in the dirty page table, dpt, by the flushing procedure. In our recovery scheme, update actions do not obtain latches on pages – instead region locks ensure that updates do not interfere with each other[1]. In addition, 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. The redo log is used as a single unifying resource to coordinate

[1]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.

the applications interaction with the recovery system, and this approach has proven very useful.

2.5 Ping-pong Checkpointing

Consistent with the terminology in main-memory databases, we use the term *checkpoint* to mean a copy of main-memory, stored on disk, and *checkpointing* refers to the action of creating a checkpoint. This terminology differs slightly from the terminology used, for example, in ARIES [MHL+92].

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. To guarantee the WAL property, a latch on the page (or possibly on the system log) is held while copying the page to disk. In our recovery scheme, we eliminate latches on pages during updates, since latching can significantly increase access costs in main-memory. It can also interfere with normal processing, as well as increase programming complexity. However, as a result it is not possible to enforce the write-ahead logging policy, since pages may be updated even as they are being written out.

For correctness, in the absence of write-ahead logging, two copies of the database image are stored on disk, and alternate checkpoints write dirty pages to alternate copies. This strategy, called *ping-pong checkpointing* (see, e.g., [SGM90b]), 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. As we shall see later, there is an I/O penalty in that dirty pages have to be written out to both checkpoints even if there was only one update on the page. However, this penalty is small for hot pages, and the benefits outweigh the I/O cost for typical main-memory database applications.

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 that were modified since the current checkpoint image was last written, namely, pages that were dirtied since the last-but-one checkpoint, are written out. 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 thus without interfering with normal operations. This *fuzzy* checkpointing is possible since physical redo log records are generated by all updates; these are used during restart recovery and their effects are idempotent. For any uncommitted update whose effects have made it to the checkpoint image, undo log records would be written out to disk after the database image has been written. This is performed by checkpointing the ATT after checkpointing the data; the checkpoint of the ATT writes out undo log records, as well as some other status information.

At the end of checkpointing, a log flush must be done before declaring the checkpoint completed (and consistent) by toggling cur_ckpt to point to the new checkpoint, for the following reason. Undo logs are deleted on transaction/operation pre-commit, which may happen before the checkpoint of the ATT. If the checkpoint completes, and the system then fails before a log flush, then the checkpoint may contain uncommitted updates for which there is no undo information. The log flush ensures that the transaction/operation has committed, and so the updates will not have to be undone (except perhaps by a compensating operation, for which undo information will be present in the log).

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 sequentially 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* [MHL+92], 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, after initializing the ATT and transaction undo logs with the ATT and undo logs stored in the most recent checkpoint, loads the database image and sets dpt to zero. As part of the checkpoint operation, the end of the system log on disk is noted before the database image is checkpointed, and becomes the "begin-recovery-point" for this 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 begin-recovery-point for the last completed checkpoint of the database are applied (appropriate pages in dpt are set to dirty for each log record). During the application of redo log records, necessary actions are taken to keep the checkpointed image of the ATT consistent with the log as it is applied. These actions 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 [Lom92]. 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.

Note that for certain uncommitted updates present in the redo log, undo log records may not have been recorded during the checkpoint – this could happen for instance when an operation executes and commits after the checkpoint, and the containing transaction has not committed. However, this is not a problem since the undo description for the operation would have been found in operation commit log records during the forward pass over the system log earlier during recovery. Any redo log records for updates performed by an operation whose commit log record is not found in the system log are ignored (since these must be due to a crash during flush and are at the tail of the system log).

3 Client-Server Recovery Scheme

Other than integration with our multi-level recovery scheme, a key feature of the client-server scheme is fine-grained concurrency control for regions. Our algorithms hinge on the simple assumption that a region is controlled by a lock, thus may easily be adapted to record-oriented or object-oriented database models. The support of fine-grained concurrency, present in our Invalidate-on-Lock scheme for cache coherency, is particularly important for distributed main-memory applications where the cost of network access due to false sharing will be proportionally higher (i.e. as compared to a few memory accesses).

In this approach, we assume a single server with access to stable storage that is responsible for coordinating all the logging, and for performing checkpoints and recovery. Multiple clients (with or without disks) are connected to the server. For simplicity of presentation, the network is assumed to be FIFO and reliable, but all our schemes can be easily modified if this is not the case. Each client and the server has its own copy of the database in main memory. A transaction executes at a single client and updates/accesses the copy of the database at the client. As a result, database pages updated by a client may not be *current* at some other client. Our scheme maintains state information at each client about each database page. A page at a client is in one of two states – *valid* or *invalid*. Invalid pages contain stale versions of certain data, and are refreshed on access by obtaining the latest copy of the page from the server.

In our client-server scheme, log records for updates generated by a transaction at a client site are stored in that site's ATT as in the centralized case. 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. Furthermore, on the occurrence of certain events (e.g., transaction commit, lock release from a site), log records in the system log are shipped by the client to the server (note that pages are shipped only from the server to clients). 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. 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 [CDF+94]). 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. The server maintains all the data structures described for the centralized case.[2] Checkpointing is performed solely at the server, and follows the same procedure as the centralized case.

Transactions follow the *callback locking* scheme [LLOW91, CFZ94] when obtaining and releasing locks. Each 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 global lock manager 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 if no transaction executing at the client is currently holding the lock.

In addition, the LLM at a client provides support for associating a point in the system log at the client with each lock; the purpose of this support will become clear later.

3.1 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. We present two variations for maintaining page state information, corresponding to "eager" versus "lazy" refresh. In both techniques, we allow two sites to concurrently update the *same page* when different locks cover different regions on the page. We begin with actions common to both methods.

- **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). It then marks the page as valid. The server keeps track of clients that have the page in a valid state.

- **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 a transaction commit, however, the log records in the system log are shipped to the server, and further actions are delayed 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.

- **Lock Release:** For each X mode region lock and operation lock that is released by a transaction, the end of the client system log is noted and stored with the lock. 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 the call-back described above, 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 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

[2]We assume there is a one-to-one mapping between ATT entries at the client sites and the server.

to the redo log for the transaction in the server's ATT. Operation/transaction commit and abort log records received from the client are processed by performing the same actions as in the centralized case when the log records were generated. The exceptions are lock release, which is driven by the client, operation commit, where the logical undo descriptor is extracted from the commit log record, and transaction commit, where 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 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.

We now 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 schemes, and both schemes follow the rule that all log records are flushed to the server before the lock which covered these updates is released 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, it is 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. Each scheme, by invalidating pages at the client, ensures that clients do not access stale data. The schemes permit regions to span multiple pages and do not require the pages spanned by a region to be known.

3.2 Invalidate-On-Update

The first invalidation scheme, based on updates, is simple, and is similar to the invalidation protocols followed in multi-processor machines in order to keep caches coherent. It is an eager protocol since a page at a client is invalidated whenever any update is made to the page at the server. The second scheme, in the next subsection, reduces these invalidation messages by tracking per-lock information at the server.

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.

For example, consider two sites updating the same page concurrently under two different region locks. Whichever site flushes its updates to the server first will cause the server to send an invalidate message to the other site, which will then re-read the page from the server. However, if this site accesses the same page again *under the same lock*, then the invalidate was not necessary, since the data in the region it has locked has not changed. The following scheme takes advantage of this observation.

3.3 Invalidate-On-Lock

The invalidate-on-lock scheme attempts to decrease unnecessary invalidations and the overhead of sending invalidation messages by associating with the lock for a region information about updates to that region. Furthermore, pages containing updated portions of a region are invalidated only when the lock on the region is obtained by a client. As a result, if two clients are updating different regions on the same page, no invalidation messages are sent to either client. Additionally, by piggybacking 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.

In the scheme, 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. This could be achieved by requiring that the lock for a region be specified by the user when the region is updated, which should be trivial since all updates must be made holding a region lock. The lock name can then be included in 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, we store 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 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. 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 locks 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 sent in any invalidate list.

4 Shared Disk Recovery Scheme

In the shared disk approach, there is no server; every site has 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 fast communication and fault tolerance. 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.

In our shared disk model, each site maintains its own copy of the database and its own system log on disk. Sites obtain locks from a 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 3. For simplicity of presentation, we assume the network is FIFO and reliable; however, the schemes can be extended if this were not the case.

We are interested in allowing multiple concurrent readers *and* writers of the same page at different sites, as long as the same region lock is not required by two sites in conflicting mode. 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.

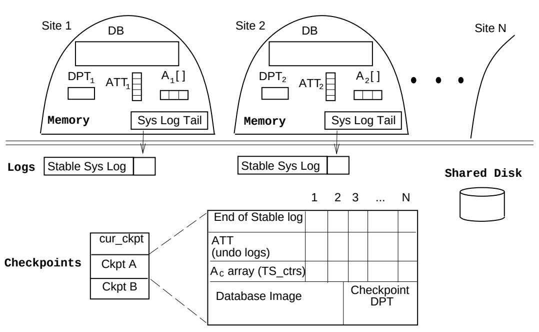

Figure 2: The Shared Disk Architecture

To solve the above problems, 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.

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. Also, in some network architectures (e.g., ethernet), the cost of sending a message to a single site may not be very different from the cost of a broadcast to all sites.

Finally, we note that although we have presented different schemes for the client-server and shared disk architectures (based on page invalidation for client-server, and based on log broadcasting for shared-disk), both schemes should be applicable to either architecture (perhaps with different performance tradeoffs, and with different requirements on concurrent updaters). For lack of space, we have not explored these alternatives here.

4.1 Data Structures

An overview of data structures used for our shared disk scheme is given in Figure 2. At each site, a global timestamp counter TS_ctr is maintained, and a timestamp obtained from this counter is stored in each physical redo log record for an update. 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.

Separate undo and redo logs are maintained for every transaction as described in the earlier schemes. Each site maintains its own version of the *dirty page table* dpt, system log, and an ATT which stores information relating to transactions that execute at that site.

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. the TS_ctr following which redo log records from the site must be applied to the database. Collectively these counters are referred to as A_C.
3. a copy of the ATT at the site (containing undo logs).

The LLM at a site stores a point in the system log with each lock as in the client-server scheme. Both the LLM and GLM also store a timestamp with each region lock, and the GLM notes which site most recently held the lock in X mode.

4.2 Normal Processing

We describe below the actions taken during normal processing to support distributed concurrency control and recovery. Recovery from system and site failure is 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.
- **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 correctness in two ways. First, similar to the client-server case, the current local end-of-log is noted when an operation or a region lock is released, 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 this could require no log flushes if the log records have already been flushed earlier due to another lock release or some other transaction's commit.

  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 bumps up its own TS_ctr to be the maximum of its current TS_ctr and the timestamp associated with the lock (received for 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.

4.3 Checkpointing

Checkpointing is initiated by a site, which coordinates the operation. The checkpointing operation consists, as for the centralized case, of three steps — 1) writing the database image by the co-ordinator, 2) writing the ATT at each site and 3) finally committing the checkpoint. 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 zero their dpts, and report their current end_of_stable_log values. Note that zeroing dpt and recording end_of_stable_log is done atomically with respect to flushes. The coordinator applies all outstanding updates, then atomically (with respect to processing further log records and flushing) records its end_of_stable_log, notes A_C from it's own A_j, and ckpt_dpt from its dpt, and then zeroes its own dpt. The coordinator then writes to the checkpoint image the ckpt_dpt, the end_of_stable_logs for each site, and the timestamp array A_C.

   Applying outstanding updates at the coordinator before noting ckpt_dpt and A_C ensures that 1) updates preceding end_of_stable_log reported by other sites have been applied to the database pages, and 2) the pages are marked dirty in ckpt_dpt 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.

2. 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).

3. Once the coordinator has written out the database image, it instructs each site to write out its ATT. Note that, as in the single site algorithm, writing the ATT at a site causes the system

log at the site to be flushed. Multiple sites can be concurrently writing out their ATTs.

4. Once every site has reported to the coordinator that its ATT has been written out, the database checkpoint is committed by toggling cur_ckpt as in the centralized case.

4.4 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. 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 and post-commit 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 set equal to the database image at site j, the dpt for each site is copied from the dpt at site j, and recovery is completed.

For lack of space we omit a proof of correctness, but a sketch of the proof is provided in the appendix.

4.5 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 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 this 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 i'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 reobtained by the lock manager at site j by consulting the GLM. These locks are all specially marked — none of these locks will be returned on call back until unmarked since they may have been held by some transactions at the local site at the time of the failure. As described in Section 2.7, rollback is performed level by level, with additional locks requested as is done during normal processing (see Section 4.2). Thus, TS_ctr at site j is bumped up 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 are unmarked once all active operations at level L_{i+1} have been rolled back. The special treatment of marked locks, along with level-by-level rollback, ensures that an in-progress operation which held a lock will in fact be protected by the lock held on behalf of the site.

5 Connection to Related Work

Multi-level recovery and variants thereof, primarily for disk-based systems, have been proposed in the literature [WHBM90, Lom92, MHL$^+$92]. Like these schemes, our schemes repeat history, generate log records during undo processing and log operation commits when undo operations complete (similar to CLRs described in [MHL$^+$92]). Also, as in [Lom92], transaction rollback at crash recovery is performed level by level. Some of the main-memory features of our scheme which impact the distributed schemes are

1. 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 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 redo log records written to the global log.

4. Our scheme uses transient undo logging which reduces the disk I/O.

In the ARIES-SD [MN91] family of schemes for recovery in the shared disk environment, each site maintains a separate log, and pages are shipped between sites. Our 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 scheme, 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 [MN91] 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 [Rah91], 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 [MN94], the authors show how the ARIES recovery algorithm described in [MHL$^+$92] can be extended to a client-server environment. In contrast to our scheme, the scheme described here 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 [MN94].

In [CFZ94], object-level as well as adaptive locking and replica management are discussed, but recovery considerations are not extensively addressed. In [FZT$^+$92], the client-server recovery scheme for the Exodus storage manager (ESM-CS) is described. This recovery scheme, based on ARIES [MHL$^+$92], requires page-level locking until end of transaction (for example, the Commit Dirty Page List).

6 Concluding Remarks

In this paper, we showed how our multi-level recovery algorithm [BPR$^+$96] can be extended to a distributed 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 scheme, suitable for a cluster of computers with a shared disk, sites participate symmetrically in transaction processing activities. We described the details of recovery after the failure of clients or the server in the client-server case and from single site and system-wide failure in the shared disk case. Our scheme allows 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 Dali Main Memory Storage Manager [JLR$^+$94]. Thus, the benefits of this algorithm are extended to the distributed schemes, including fuzzy, dirty-page only checkpointing, reliance on the log for functions which are typically page based, low overhead logging with undo records written only due to a checkpoint, and per-transaction logs for low contention.

We plan to explore the performance of these schemes through experimentation, and then build a distributed, data-shipping version of Dali based on these algorithms.

References

[BPR+96] 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.

[CDF+94] 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.

[CFZ94] 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.

[DKO+84] 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.

[FZT+92] 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.

[GMS92] 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.

[Hag86] Robert B. Hagmann. A crash recovery scheme for a memory-resident database system. *IEEE Transactions on Computers*, C-35(9):839–847, September 1986.

[JLR+94] 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.

[JSS93] H.V. Jagadish, Avi Silberschatz, and S. Sudarshan. Recovering from main-memory lapses. In *Procs. of the International Conf. on Very Large Databases*, 1993.

[LLOW91] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The objectstore database system. *Communications of the ACM*, 34(10), October 1991.

[Lom92] 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.

[LSC92] 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.

[MHL+92] 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.

[MN91] 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.

[MN94] 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.

[Rah91] 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.

[SGM90a] 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.

[SGM90b] 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.

[WHBM90] 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.

A Correctness of Shared Disk Algorithms

The basic idea behind the proof of correctness is to treat the combined system logs conceptually as a single log, merged according to the timestamps. The checkpointed timestamp array A_C is essentially a pointer into this logical log, and constitutes the logical log restart recovery point. We show correctness of the shared disk recovery and cache coherency algorithms by showing the following:

1. For every update written out during the checkpoint operation, and that had not committed before the end of checkpointing, the undo log record describing the update is also written out.
2. All updates described by log records before the logical log restart point (array A_C) noted in the checkpoint have made it to the database image.
3. History is repeated as a consequence of applying the redo log records during restart recovery.

Point 1 is ensured since the ATTs are checkpointed after an update completed, and every system log is flushed to disk before the checkpoint completes, so that all pre-committed updates get committed. Thus, the undo log for any uncommitted update is guaranteed to be written to disk.

Point 2 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).

Point 3 is ensured due to the following reasons –

1. All physical log records are applied during recovery in timestamp order – immediate from the recovery algorithm.
2. For a given region, the order of log record timestamps reflects the order of updates which generated the log records. For every log record, L, (in the system log of a site) describing an update, the log record, L' for the preceding (conflicting) update is also in some site's stable log with timestamp less than the timestamp for this log record. The reason for this is that before a region lock is released by a site, updates covered by the region lock are appended to the system log, flushed to disk, and broadcast to the network. TS_ctr at the receiving site is bumped up and so must be larger than the timestamp contained in L' when log record L is moved to the system log and assigned a timestamp.
3. If a timestamp contained in a log record for site i is less than or equal to $A_j[i]$, then the log record's effects must have made it to the copy of the database at site j.
4. Finally, we show that if a log record, L1, from site i is applied to a page during recovery, then a conflicting log record, L2, from another site, j, with timestamp higher than the L1's timestamp, will also be applied. In other words, the timestamp of the second log record is greater than $A_C[j]$.

   Suppose log record L1 is applied during recovery, and it describes an update at site i. Suppose further that the update for L1 precedes another update at site j, described by L2. Then, at the coordinator site for the last completed checkpoint, L2's timestamp is larger than the timestamp array entry for j. The reason for this is that L1 is first broadcast before locks are released, and only later is L2 broadcast to all the sites. Since L1 is applied, its timestamp must be greater than $A_C[i]$, which means the broadcast of L1 did not reach the last site that did the checkpoint. But then neither could the broadcast of L2 – so the timestamp $A_C[j]$ must be less than the timestamp of L2, and L2 would be executed as well.