Recovery for Shared Disk Systems Using Multiple Redo Logs

David B. Lomet

Digital Equipment Corporation
Cambridge Research Lab

CRL 90/4 October 1,1990

Abstract

A new method for redo logging and recovery is described. It is designed to work in a data sharing system where multiple nodes can access common data. Particularly important is that each node can have its own log. Crash recovery is possible based on the contents of just one of these logs. Media recovery is supported via a form of merging of the logs. The method requires no time synchronization between nodes, and does not use timestamps to order the log records during the merge. The method should work with many undo recovery methods. Conventional checkpointing schemes can be adapted to the scheme, enabling the redo scan point to be advanced and the log to be truncated. Finally, the method makes possible a new paradigm for distributed DBMSs which has the potential to exploit inexpensive desktop processing and improve availability and responsiveness.

Keywords: logging, database recovery, data sharing systems
©Digital Equipment Corporation 1990. All rights reserved.

1 Introduction

1.1 Data Sharing Systems

There has been little written in the open literature on how to handle logging in system configurations where a number of computers all access a collection of shared disks. This configuration is called a cluster[6] or a "shared disk system"[2,13]. A database system that exploits the configuration such that any computer can access any of the data within a single transaction is called a "data sharing" system. Two commercial systems offering data sharing are Digital's Rdb/VMS (and DBMS) [10] and IBM's IMS/VS Data Sharing product[14].

A data sharing system does what is called "data shipping". That is, a data block, as it comes from the disk, is sent to the requesting computer. In contrast, in a "function shipping" system, a collection of operations is shipped to a single computer designated as the server for the data. This later system is also called a "partitioned" system[11,12]. The distinguishing feature is this. With data sharing, a data item can reside in the caches of multiple computers [nodes], and be updatable from these nodes. With a partitioned system, or a centralized [single node] system, a data item can reside in at most one cache.

1.2 The Traditional Advantages of Data Sharing

Data sharing database systems have traditionally had three potential advantages compared with partitioned systems.

1.2.1 Performance in Some Applications

In a partitioned system, any transaction that touches data in more than one partition will be a distributed transaction. That same transaction, in a data sharing system, may well run entirely on one node. These transactions, involving data either in a local partition or the shared data, need not be "distributed" in a data sharing system. Hence they do not need to involve a multi-phase commit protocol with the messages that this entails. For relatively simple transactions in which the distributed concurrency control overhead is small, a data sharing system may well out-perform a partitioned system.

Example:

```
A Debit/Credit transaction may perform better on a data sharing system, where
the teller and branch relations are partitioned but the account relation is
shared.  A "distributed" transaction is not needed, even when a customer
conducts his business outside of his home branch, since the account records are
shared.  The cost of coordinating access to the account relation via distributed
locking is an added expense.  But only a single DISTRIBUTED lock is required for
Debit/Credit, i.e. for the account record.  In the partitioned system, messages
are needed to fetch the account record when it is remote, then additional rounds
of messages are needed to coordinate the commit.
```

1.2.2 Dynamically Scalable Performance and Load Balancing

Partitioning a database has been used successfully to achieve scalable performance[15]. However, partitioning requires advanced planning and is difficult to change in a dynamic way in response to changes in load. With data sharing, since any processor in the cluster can access the shared data, if the load becomes unbalanced, a processor can readily be switched over to help out. Further, as the system grows, re-partitioning is not needed when new processors are added to the system. The new processors simply are added to the cluster and

pick up their share of the work. (Note that data must still be distributed with some care over the disks of the system so as to balance the load on the disks.)

1.2.3 Fault Tolerance

The same flexibility that permits performance to scale, by dynamically adjusting the load, provides a very powerful fault tolerance capability. A processor failure is but an extreme example of the need for dynamic re-balancing of the load, in this case switching it from the failed processor to one that remains available. Partitioned systems can also provide this fault tolerance, but again, it must be carefully planned ahead of time. And it is typically more rigid in that (1) frequently only one error can be tolerated and (2) it is much harder to balance the load from the failed processor.

1.3 The Log Problem

Log-based methods[1,3,5] have been the preferred approach for providing recovery in database systems. And, when handling logs, partitioned systems have had a distinct advantage compared with data sharing systems. This advantage negates much of the potential performance advantage discussed above for data sharing systems in some applications.

Partitioned systems share with centralized systems the property that log records for a data block are recorded on one and only one log. This property permits recovery for a data item based solely on the contents of one log. With partitioning, this one log is immediately accessible to the processor supporting the partition, i.e. it is private to the partition and is under strictly local control. No distributed system coordination is required.

Data sharing systems can also use only a single log. But for, them, this can have a much more substantial cost. Two techniques can be employed.

1. Ship the log records to a log server that is responsible for recording recovery information for the data. Such "remote" logging is very expensive of system resources, involving both messages and I/O writes. The delay involved in waiting for an acknowledgement from the logging processor can be substantial, reducing concurrency as well as increasing response time.
2. Synchronize the use of a common log, in effect taking turns writing to this common log. This too is expensive, involving extra messages for the coordination.

1.4 Our Method: Private Redo Logs

Our goal is to support data sharing while permitting each node that shares the data to have its own local log that it uses to log all changes that it makes to data wherever that data is located. We call such a system a "shared data/private log" system. A private log avoids both the sending of remote logging messages and synchronization for use of a common log.

Digital's Rdb/VMS system[10] already has a private log for transaction recovery. And for transaction recovery, no synchronization is needed for logging. However it is an UNDO log and Rdb/VMS recovery requires forcing all updated pages back to disk at transaction commit. Higher performance is possible using a REDO strategy exactly because pages need not be forced to disk at transaction commit. This permits many transactions to execute and change a collection of pages, and the pages need only be written to disk occasionally, and not in a forced manner.

What we describe in the remainder of the paper is a REDO logging scheme in which shared data items can have their REDO recovery information stored on any of several logs. When these logs are organized so that each node has access to a private log, coordinating the use of the log becomes strictly a matter for the node. Distributed synchronization for log usage is unnecessary.

There are many ways of performing recovery and of handling the REDO phase of it. The technique that enables multiple logs to contain recovery information for a single data item can be used with many of these methods. To show how the technique works, we describe it in the context of the "repeating history" technique [7], which we like because of its functionality and simplicity. In particular, it permits operation logging and our method can exploit that capability.

REDO is only part of system recovery. Most systems will also require an UNDO recovery capability as well. Further, systems will usually require "log management" so as to determine where to start the redo scan and how to truncate the log. Since our technique can work with several strategies, we do not pursue these issues in any depth, pointing out only the places where care must be taken when they interact with REDO recovery.

The new capabilities that are enabled by our method permit the use of multiple logs for redo recovery such that

1. each node can perform crash recovery using only its own log, without synchronizing with other nodes in the system;
2. media recovery is possible via a "merging" of log records from multiple logs;
3. the above features can be accomplished without a NODE performing any time synchronization with any other NODE and without the use of timestamps in log records. Note that synchronization and timestamps are required in [9].

1.5 Organization of the Paper

Section two provides background information for the remainder of the paper. Terms are defined and the fundamental conditions that need to be satisfied in order to permit effective use of private logs are discussed. Finally, how state identifiers are used to control the redoing of operations is described.

Section three describes the role of state identifiers in a multi-log system. The identifier for the before state of the data item being updated is stored in the log record for each operation. Further, the state identifiers are ordered so as to facilitate log merging. In section four, we discuss how to manage state identifiers so as to preserve their uniqueness and "seamless" ordering. These properties must survive undo recovery and the allocation and freeing of the data.

Section five describes how the system operates when using multiple private logs. Log format and management is discussed. We describe how the system prepares for possible crash recovery during normal operations, including the logging for the various activities of a transaction. We then describe how recovery is performed, with the usual three phases, analysis, redo, and undo.

We describe, in section six, how data sharing with private logs opens up the possibility of organizing distributed database systems in a new way that we call the "lending library" model. This model has some very important advantages over current distributed databases.

We end the paper with a brief discussion section.

2 Background

2.1 Definitions of Terms

We begin by defining terms that will be used extensively in this paper.

- Data block[BLOCK]: BLOCKS are the recoverable objects of the system. A BLOCK is an integral number of pages of persistent storage. A BLOCK is identified by a BLOCK identifier or BID.
- Database Cache[CACHE]: the volatile system storage that contains BLOCKs. BLOCKs can be operated on only when in CACHE. The contents of the CACHE may be lost during system crashes.
- Persistent storage[PSTORE]: the system storage whose contents are presumed to persist when part or all of the system crashes. Traditionally, this storage has been magnetic disk.
- Archive storage[ASTORE]: the system storage that is used to store information that permits reconstruction of the contents of PSTORE should the PSTORE database become unreadable. ASTORE is frequently magnetic tape, but could also be magnetic or optical[even WORM] disk.
- Transaction Log[TLOG]: a sequential file used to record information that permits an operation of a transaction to be performed again[redone], should the system crash prior to committed data being written to PSTORE. Recovery involving the TLOG assures the durability of transaction effects in PSTORE.
- Media Log[MLOG]: a sequential file used to store log records for sufficient duration so as to provide media recovery, i.e. recovery from failures of the PSTORE medium. Typically, the MLOG is the unique part used solely for media recovery, to which is appended the TLOG, which will complete the restoration. The TLOG information forms the basis for the MLOG, i.e. is the source of information from which the MLOG is generated.
- State Identifier[SI]: a unique value for a BLOCK that identifies the state of the BLOCK at some particular time, before or after some operation upon the BLOCK.
- Log Sequence Number[LSN]: the address or relative position of a log record in a log. LSNs traditionally have been used as state identifiers.
- Node[NODE]: a computer which supports one CACHE in a single shared memory. The computer may have multiple processors, as with tightly coupled multiprocessors, or a single processor.

2.2 Characterizing BLOCKs by Their Recovery Needs

We wish to characterize the versions of BLOCKs that may be available after a crash, in terms of how many of the logged actions on how many logs are needed to make the available version current. This has obvious impact with respect to how extensive or localized recovery activity will be.

We distinguish three kinds of BLOCKs

1. CURRENT: A version of a BLOCK is CURRENT if all updates that have been performed on the BLOCK are reflected in the version. A BLOCK having a CURRENT version after a failure does not need any recovery. When dealing with arbitrary system failures, one cannot ensure that all BLOCKs are CURRENT for redo without always "writing-thru" the CACHE to PSTORE whenever an update occurs, which is expensive. And BLOCKs cannot be guaranteed to be CURRENT for both redo and undo unless shadowing is used.
2. ONE-LOG: A version of a BLOCK is ONE-LOG for redo if only one node's log can have updates that have not yet been applied to it. It is ONE-LOG for undo if only the results of incomplete transactions on a single log need to be undone. When a failure occurs, at most one node need be involved in a given phase of recovery. This is desirable as it avoids potentially extensive coordination during recovery, as well as additional implementation cost.
3. N-LOG: At the cost of more expensive and complex recovery, one can avoid the overhead involved in assuring that BLOCKs are at worst ONE-LOG. More than one log will have records that need to be applied to the available version of the BLOCK should a failure occur.

It is possible for a BLOCK to be ONE-LOG with respect to redo recovery and be N-LOG with respect to undo recovery. Ensuring ONE-LOG recovery is discussed in the next subsection.

2.3 Node Recovery Isolation

Without care, at the time of a crash, some BLOCKs will be N-LOG. Below, we show how to guarantee that a BLOCK will be ONE-LOG. Both redo and undo recovery are treated.

2.3.1 Isolated Redo Recovery

N-LOG BLOCKs require coordination between nodes for their recovery. Such coordination is possible. Their updates were originally sequenced during normal system operation using distributed concurrency control. However, it is desirable to avoid the overhead of concurrency control during recovery. Further, the concurrency control required here is more stringent than normal concurrency control, as the sequence of actions on each BLOCK is completely prescribed during redo since we are "repeating history".

ONE-LOG Redo Requirement: BLOCKs can be made ONE-LOG with respect to redo recovery if dirty BLOCKs are written to PSTORE before the TLOG upon which their recovery information is written changes.

When each CACHE uses a separate TLOG, the above requires writing dirty BLOCKs to PSTORE whenever a BLOCK moves between CACHEs. A requesting node then gets a clean BLOCK whenever it enters the node's CACHE. During recovery, only the records on the TLOG of the last node to change the BLOCK need be redone for the BLOCK. All other actions of other nodes using other TLOGs have already been captured in the state of the BLOCK in PSTORE.

2.3.2 Isolated Undo Recovery

For N-LOG undo, multiple transactions, logged on multiple logs, can have uncommitted data in a BLOCK simultaneously. A system crash would require these transactions to all be undone. This may require, for example, locking during undo recovery to coordinate BLOCK accesses. Note here, however, that the sequencing of the undo operations is not prescribed. Hence, concurrency control may be applied during undo recovery, in exactly the way that it is used during normal system operation.

ONE-LOG Undo Requirement: BLOCKs can be made ONE-LOG with respect to undo recovery if dirty BLOCKs containing uncommitted data are not permitted to change the TLOG upon which their recovery information is written.

A data sharing system typically executes a transaction upon shared data at only a single NODE with its own CACHE. When each CACHE uses a separate TLOG, that means that we can enforce the above requirement by preventing a transaction on a second node from updating the same BLOCK concurrently. This can be achieved through having a lock granularity that is no smaller than a BLOCK. A requesting NODE then receives a BLOCK in which no undo processing by another node is ever required. If a transaction from another NODE had updated a BLOCK and then aborted, the effect of that transaction will have already been undone.

Though ONE-LOG undo reduces complexity, the impact on system performance of N-LOG undo at recovery time is much less than for N-LOG redo. Only the small set of transactions that were uncommitted at the time of system crash need undoing. We believe that N-LOG undo is the most likely choice here because it permits arbitrary locking granularity and because relatively few transactions need undo compared with the number that need redo.

2.4 Independent Recovery

We would like to make explicit an assumption that is frequently made but rarely never stated concerning redo recovery. We call this assumption the independent redo recovery assumption. (The first published paper of which we are aware that makes this explicit is [7].)

Independent Redo Recovery Assumption: The state of a BLOCK can be transformed to a successor state during redo recovery based solely on the state of the BLOCK at the time the original operation was applied and the contents of the log record that describes the operation.

This assumption is important because it means that recovery of the database can be done separately for each BLOCK if we can find the log records that apply to the states of each BLOCK that we have available at the time of a crash. The assumption may seem a natural one to make but making it explicit provides an explanation for why certain information is logged that may not seem strictly necessary.

Note that when the entire BLOCK state resulting from some action is stored in the log record that the independent recovery assumption is trivially satisfied. In fact, the independent recovery assumption is easily seen to be true when redo actions install, idempotently, a complete or partial successor state that was explicitly recorded in the log record. Things become more interesting when a recovery system does operation logging and recovery is accomplished by re-executing the original operation during recovery. For these systems, it

is important to eliminate inter-BLOCK dependencies by storing the state information from the other BLOCKs that is needed by the operation in the log record.

Example:

When a B-tree node is split, a new node is created that is initialized to contain approximately half of the contents of the old node. Assume that only the old node's identity, some way of identifying the state that this node was in at the time of the split, the split key value, and that the operation is a node split are logged. This information is logically sufficient. However, it requires the recreation of the state of the old node at the time of the split in order to recover the state of the new node resulting from the split. This violates the independent redo assumption. To guarantee independent redo recovery, the entire initial contents of the new node must be recorded in the log record for the split. Recovery for the new node is then possible without any dependencies on other nodes, based solely on the contents of log records for the new node.

2.5 BLOCK State Identifiers

2.5.1 The role of State Identifiers

A log record must be applied to the BLOCK when the recorded state of the BLOCK is appropriate for the action designated by the log record. A sufficient condition for correct redo, given the independent recovery assumption, is to apply a logged action to a BLOCK when the BLOCK is in the same state as it was when the original action was performed. Note here that the independent redo assumption means that other BLOCKs need not be in the state they were in when this action was performed. Then, assuming that the original action was correct, the redone action will also be correct. Correct redo then will not require idempotent redo operations. Hence, operation logging is supported.

We do not wish to store the entire contents of a BLOCK state on the log. The usual way of handling this is to create a proxy value or identifier for the BLOCK state. This "state identifier"(SI) is much smaller than the complete state and can be inexpensively used in its place, so long as one can re-create the full state when necessary. A state identifier is defined by storing a particular SI value, called the "defining state identifier" or DSI, in the BLOCK. The DSI denotes the BLOCK state in which it is included.

State re-creation is done by accessing the entire BLOCK state in PSTORE (or ASTORE) during recovery and noting its DSI. This BLOCK state is then brought up to date by applying logged actions as appropriate. Knowing whether a log record applies to a BLOCK involves being able to determine, from the log record, to what state the logged action applies.

One can express the role of state identifiers in redo recovery, given the independent recovery assumption, as follows. Let REDO be the operation of redoing a single operation involving a BLOCK. Then

$$REDO(BLOCK(BID, SI), LOG(BID, SI, Op, Data)) -> BLOCK(BID, Succr(SI))$$

where $LOG(BID, SI, O\ P, DATA)$ is the log record describing the action that was originally executed to transform the BLOCK identified by BID in state SI into its successor state, $Succr(SI)$. The action described involves executing the operation OP on $DATA$ and the BLOCK BID when it was in state SI. [One can assure that the original execution of OP, and its REDO are precisely the same by using $REDO$ to execute OP initially.] Most of the rest of this paper can be interpreted as exploring techniques for finding and identifying the

log record $LOG(BID, SI, OP, DATA)$ and for making sure that all such needed log records are stably stored so as to be accessible across system crashes.

2.5.2 Log Sequence Numbers as State Identifiers

In a centralized or partitioned system, the physical sequencing of records on the single log is used to correctly order the redoing of actions. That is, action B on a BLOCK immediately follows action A on the BLOCK, and applies to the state created by A, if B is the first action on the log that follows A and that applies to the same BLOCK. So, if action A has been redone, the next log record to apply to the BLOCK will be action B.

A BLOCK's DSI determines when to begin applying log records to that state. With single log systems, log sequence numbers can identify BLOCK states. Hence, most systems use LSNs as SIs[3,5,7]. The LSN that serves as DSI for the BLOCK identifies the last record in log sequence to have its effect reflected in the BLOCK. We call this the BLOCK LSN.

A redo log record has an LSN implicitly associated with it, i.e. the LSN is the pointer to the log record. This LSN also names the state produced by the logged action. Thus, the LSN plays the role of an "after state identifier" or ASI. A log record applies to a state of a BLOCK if its LSN is *min{LSN | LSN > BLOCK LSN}* of those log records for the BLOCK. That is, it represents the next operation performed after the one named via the BLOCK LSN.

3 State Identifiers For Multi-log Data Sharing Systems

3.1 ONE-LOG Redo Recovery

Our goal is for all BLOCKs to be ONE-LOG for redo recovery from system crashes. We accomplish this by simply adopting the strategy of writing BLOCKs back to PSTORE whenever the TLOG on which their log records are written changes. Thus, redo crash recovery will not require distributed concurrency control. However, multiple TLOGs may contain records for a block even though only one node's records are applicable to the version of the BLOCK in PSTORE. The problem then is how a NODE determines that its logged actions are the ones to be applied to a BLOCK.

A major advantage of using LSNs as state identifiers is that it is possible to determine the LSN associated with a log record without actually storing any data in the log record since the LSN is the location of the record in the log. However, LSNs pointing to local log records are not sufficient to serve as unique state identifiers when there are multiple logs. It seems that we must explicitly store system wide state identifiers in each log record.

One could augment an LSN with a log identifier or LI, that indicates on which log the log record so named resides. Thus, state identifiers might be [LI,LSN] pairs. This can be made sufficient for ONE-LOG recovery(see below). However, it is highly desirable for SIs to be totally ordered when treating N-LOG recovery such that the ordering agrees with the time sequence of the actions logged. [LI,LSN] pairs lack such an ordering.

Ordered SIs are exploited in an essential way for ONE-LOG recovery in [9]. There, the state identifier associated with a log record is an ASI, which is the same role played by an LSN. These state identifiers are required to be monotonically increasing values, effectively update sequence numbers. The test of whether a log record applies to a BLOCK in some state is whether this ASI is the first one greater than the BLOCK's DSI, exactly as with LSNs. Importantly, this technique did not make it possible to derive the SI for the BLOCK

state before the operation is executed from the ASI. While the technique is sufficient for ONE-LOG recovery, it requires extra mechanism to permit convenient N-LOG recovery. In [9], timestamping is used to supplement the SIs.

3.1.1 Before State Identifiers

In our technique, we include, in each log record, the precise identity of the BLOCK state that is seen BEFORE we perform a logged action. This is the "before state identifier" or BSI. When a log is read, we know whether a log record applies to a BLOCK state by testing for equality between the log record BSI and the BLOCK DSI.

Storing a BSI in the log record is a very powerful technique. It makes it possible to identify which log record to apply to a BLOCK in some state independently of log ordering or the number of logs. Log records in an unstructured heap can still provide recovery, and without having to sort the records. One does not have to scan the logs in some order to establish whether a log record applies to a BLOCK in some state. This is sufficient to ensure recovery, however, only with the independent redo recovery assumption being true. Actions might not be redone in the original sequence for collections of BLOCKs. Only correct sequencing for a single BLOCK is assured.

We must also be able to determine the ASI for a BLOCK after applying the log record so that we can update the DSI and prepare for applying the next operation. Storing the ASI in the log record is one way of accomplishing this. Thus, each log record could include <BSI, ASI, Operation> and this completely describe the state transition for the BLOCK. However, if we can derive the ASI from the BSI, or from the location of the log record, i.e., its [LI,LSN] pair, we will not have to store an ASI in log records. The same derivation must, of course, be used during recovery as is used during normal operation. [There is a potential symmetry here, of course, in that if we can derive the BSI, we could as easily store only the ASI in the log record. While LSN variants can be used as ASIs, they cannot be used as BSIs since they are not known until the execution of an operation. Further, one cannot derive a BSI from an LSN used as an ASI.]

Given the uniqueness of SIs and the ONE-LOG condition, during crash recovery, only one log will contain a log record with a BSI that matches a BLOCK's DSI. For media recovery, which is N-LOG, not ONE-LOG, the advantage of using BSIs in log records is even more important. In particular, BSIs help us in the "merge" of multiple logs.

3.2 N-LOG Redo Recovery

We form a separate media log[MLOG] from the truncated parts of each TLOG. These are the parts of the TLOGs that are no longer needed to bring the PSTORE versions of BLOCKs up to current versions. However, these records are still needed should the PSTORE version of a BLOCK become unavailable and need recovery from the version of the BLOCK on ASTORE. Hence, we have multiple MLOGs, one for each TLOG.

Media recovery has many of the same characteristics as crash recovery. There needs to be a stably stored version against which log records are applied. There are important differences in detail however.

- the stable version of the BLOCK against which the media log records are applied is the version last posted to ASTORE;

- the MLOG needs to persist for intervals coordinated with the posting of BLOCKs to ASTORE;

Particularly important is that restoring BLOCKs from ASTORE involves N-LOG redo recovery. We do not write BLOCKs to ASTORE every time they move between CACHEs because this is too expensive. Because we store BSIs in log records and test for BSI-DSI equality, we conceptually have enough information to provide N-LOG recovery without further ado. We simply look for the log record for a BLOCK whose BSI matches the DSI stored in the BLOCK. However, managing recovery is a "nightmare" unless we can "merge" the logs. This avoids searching "high and low" for applicable log records.

3.2.1 Log Merging

When performing N-LOG recovery where BSIs are stored in log records, it is desirable to efficiently "merge" the multiple logs to perform recovery. The merge need not be based on a total ordering among log records, but only on the partial ordering that results from the ordering of log records for the same BLOCK. At times there can be multiple logs that have records whose actions can be applied to their respective BLOCKs. Given the independent redo assumption, it is immaterial which of these actions is applied first during media recovery since each BLOCK's recovery can proceed separately.

For N-LOG recovery, we would like to merge the multiple logs and apply their records in a single pass. Ordering SIs and knowing the successor function for SIs makes this easy. We assume that a single NODE performs the merge and that SIs are ordered and have known successors. Start with any log, and read its first log record in the redo scan. (Determining this record is a function of log management and is treated in section five.) Then the log merging proceeds as follows:

1. If the log record's BSI is less than the BLOCK's DSI, then the logged action is already incorporated into the BLOCK. Redo is not needed. Skip over the record. Continue processing the current log.
2. If the log record's BSI is equal to the BLOCK's DSI, then the logged action applies to the BLOCK. Redo the logged action and generate the ASI for the action. Post this ASI as the BLOCK's DSI. Continue processing the current log.
3. If the log record's BSI is greater than the BLOCK's DSI, then there are additional actions on other logs that need to be applied to the BLOCK before this action is redone. Pause in the reading of this log. Read the next record from another, non-paused, log.

Methods that use ASI's in log records [and cannot derive the BSI] have not distinguished case 2 from case 3. When a log record is encountered with an ASI greater than a BLOCK's DSI, it is not known whether there are intervening actions on other logs that need to be applied prior to applying this log record. That is because it has not been possible to uniquely determine, the BSI from the ASI. This is why [9] supplements the state identifier in the log record with a timestamp, and further requires that the clocks from which the timestamps are derived be synchronized. Timestamp ordering is used to control the log merge, not the state identifiers.

There is no ambiguity in our method. Case 3 REQUIRES that there be at least one other log that contains records for the BLOCK that precede the current one. A paused log with a waiting log record is simply regarded as an input stream whose first item(in an ordered sequence) compares later than the items in the other input streams[logs]. Processing continues using the other logs until they are paused. It must be the case that the current

record of the paused log can be applied to the BLOCK at some future time, since the BSI would not be greater than the BLOCK's DSI without intervening updates on other logs. When this occurs, "unpause" the log.

As long as there are no missing SIs in the BSI/ASI sequencing for each BLOCK, all of the logs will never be simultaneously paused. Log records are recorded in each log in increasing time order. Hence, a log record at the front of a log was recorded earlier than any records that come after it. Further, if two records apply to the same BLOCK, the one with the smaller BSI was written earlier in time than the one with the larger BSI. If all logs are simultaneously paused, this implies a circular waiting among the logs in which each paused log has an earliest record that was written only after records on another log. And this circular waiting implies a circularity in the time ordering of at least some of the log records, which is impossible. Thus, a merge of the logs is always possible.

4 Seamless Ordering of State Identifiers

For normal operations on allocated BLOCKs, it is easy to order SIs and to provide a derivation of ASI from BSI. When a BLOCK is first allocated, it can be given an SI of zero. Then, a simple incrementing scheme can be used where $ASI = BSI + 1$. This means that SIs are effectively update sequence numbers. There is no need for any further cleverness here.

The argument that not all logs will be paused simultaneously, and hence that all log records will eventually be applied, required the assumption that there were no gaps in the BSI/ASI sequencing of log records for each BLOCK. Hence, we need to constrain our logging protocol so as to ensure that no log record whose action has effects included in the stable BLOCK state is omitted from the stable log. While it is permissible to drop the log tails whose effects were never stably recorded or committed, our N-LOG merge protocol tolerates no gaps in the ASI/BSI sequencing. A gap in the SIs results in a permanently paused log during our merge. Hence the SIs of a BLOCK must be "seamlessly" ordered.

4.1 Forcing Logs when BLOCKs Change Logs

We enforce the Write-Ahead Log protocol for TLOGs, even when a log record only contains redo information. Were a BLOCK to be written to PSTORE prior to the log record for the last update for the BLOCK, then the following sequence of events could occur:

1. A BLOCK containing uncommitted updates is written to PSTORE at one NODE. Its last update is not forced to the NODE's TLOG.
2. A second transaction on another NODE further updates the BLOCK and commits. Its SIs for the BLOCK increment the DSI that it observed in the BLOCK. At its moment of commit, its logged actions are forced to its TLOG. However, since it was on a different NODE, the writing of these log records do not assure that the uncommitted transaction on the original node has its log record for the BLOCK written.
3. The original NODE now crashes. In particular, the log record for the uncommitted transaction is never written to its TLOG. This creates a gap in the ASI/BSI sequencing for the BLOCK because of this missing log record.
4. Should the BLOCK in PSTORE become unavailable, e.g. via a disk failure, media recovery via MLOG merge would fail because the MLOG merge would permanently pause the MLOG containing the BSI just after the missing and unlogged action.

The WAL protocol is a necessary condition for an unbroken sequence of logged actions. It is also sufficient when blocks are ONE-LOG. Only a BLOCK moving from one NODE's CACHE to another can result in the commit of a later transaction forcing TLOG records for the BLOCK without the forcing of TLOG records for all prior updates to the BLOCK. Our ONE-LOG redo recovery assumption guarantees that all BLOCKs moving between CACHEs must be written first to PSTORE. This writing to PSTORE causes the WAL protocol to force write to the TLOG of the original NODE all records through the log record for the last update to this BLOCK.

For N-LOG BLOCKs, the WAL protocol is not sufficient. We need to force the TLOG whenever a BLOCK moves between CACHEs, even if the BLOCK is N-LOG and is not written to PSTORE. Without this force, the seamless sequencing of SIs is not guaranteed, as the example above makes clear.

4.2 The Interaction with Undo Recovery

The redo technique that we are describing can work well with several methods of handling undo. Here we are focusing on the redo phase, understanding that complete recovery also obviously requires an undo phase as well. What we need to understand is what requirements the undo phase of recovery must satisfy so as to permit redo recovery to succeed.

It is important that undo recovery preserve the chain of BSI/ASI state changes recorded in the log. In particular, it is important to include on the log the state change(s) that transforms a BLOCK from a state in which it contains uncommitted updates to a state in which the uncommitted updates have been removed. This preserves the uniqueness of state identifiers and permits redo recovery to remain ONE-LOG even when combined with quite different undo recovery techniques.

The log records of actions that record the undoing of other actions are called compensation log records or CLRs [3,5,7]. CLRs are effectively undo information that is recorded on the TLOG to enable the effects of operations to be undone as part of the redo phase of recovery. While it is conceptually possible to avoid writing CLRs, the recovery method is much simplified with them. In particular, this is the case for N-LOG recovery via log merging.

4.3 Allocation and Freeing of BLOCKs

When a BLOCK has been freed, and then is re-allocated, we must not re-assign it an SI of zero, as this would result in non-unique state identifiers. Several log records, each with the same BSI, might appear to apply to a BLOCK, but only one would be correct. We must ensure that the SI numbering used in the prior allocation continues uninterrupted in the new allocation. Hence, the BSI for a new BLOCK is the ASI of the BLOCK as it is freed.

Uninterrupted SI numbering can be achieved by reading the DSI stored in the BLOCK as a result of the last update operation of its prior incarnation, when the BLOCK is re-allocated. Then, normal SI incrementing can be continued. The problem with this solution is the need to read newly allocated BLOCKs before using them. And this may require a read from PSTORE. Because we would like space management to be done efficiently, with a minimum of I/O activity, we wish to avoid the read before allocation penalty.

Systems must stably bookkeep free space in PSTORE. There is normally a collection of space management BLOCKs (which we call SMBLOCKs) where this information is recorded. It is useful to think about the allocation and freeing of BLOCKs, not as operations on the managed BLOCKs, but rather as (update) operations on SMBLOCKs. Hence, allocate and free operations do not update the BLOCKs being allocated or freed, and hence do not increment those BLOCKs' DSIs. Rather, they update the SMBLOCKs and increment their DSIs. This view permits SMBLOCKs to be recovered using the same techniques as for all other BLOCKs.

To provide seamless SI numbering across allocate and free operations, an initial SI is stored with each free BLOCK as part of its space management information in an SMBLOCK. For BLOCKs not previously allocated, the initial SIs are zero. For previously used BLOCKs, the initial SIs are the ASIs of the last update operations on the BLOCKs prior to their being freed. On re-allocation, initial SIs become the BSIs for first updates on newly allocated BLOCKs. By recording initial SIs for BLOCKs in the SMBLOCKs, reading BLOCKs from PSTORE in order to re-allocate them is avoided.

One NODE cannot re-allocate a BLOCK freed by another NODE until the freed BLOCK's existence is made known to it via reading the appropriate SMBLOCK. SMBLOCKs are managed like ordinary BLOCKs. That is, they are written to PSTORE whenever they move from one NODE's cache to another, and may need to be read from PSTORE. However, no BLOCK is read as part of its re-allocation. Hence, no additional I/O activity is needed to provide seamless SI numbering. Initial SIs need only be added to the space management information already maintained.

Storing initial SIs for free BLOCKs may substantially increase the amount of space management information. Our ability to minimize this space penalty relies on the observation that most free space has never been allocated. Hence it will have a ZERO initial SI, which needn't be represented even once. The previously used free space will be small, as databases are usually growing. Hence, we store initial SIs individually only for the previously used BLOCKs. Given our observation, the increased storage needed should be small.

5 System Operation

In this section we examine the overall operation of recovery in a data sharing database system that exploits multiple redo logs. Much of this is common to many recoverable databases. Our purpose here is to show how our multiple log techniques fit into conventional approaches. Hence, we highlight this interaction below.

5.1 Format for Log Records

The structure of each of the redo logs is conventional, with the exception that we include a before state identifier in the record. We omit any mention of how or whether undo information is logged except to provide for compensation log records that permit the BSI/ASI sequencing of log records to be seamless. Such undo information does, however, commonly exist in redo/undo recovery schemes.

The record's redo information can be formatted as follows:

```
---------------------------
|TYPE|TID|BSI|BID|OP|DATA  | [LSN implicit]
---------------------------
```

The attributes of a redo record are as follows:

- **TYPE** identifies the kind of log record this is. There are several types, including redo record, compensation log record, commit related records, and checkpoint records, all stored on the log.
- **TID** is a unique identifier for the transaction. It may help us find the undo record corresponding to this redo record. It also helps us determine whether the action needs redo and whether it eventually may need undo.
- **BSI** is the before state identifier, as described previously.
- **BID** identifies the BLOCK modified by the action logged with this record.
- **OP** indicates the operation that needs to be executed in order for recovery to be accomplished.
- **DATA** provides enough information for the action to be redone without reference to the states of other BLOCKs. This is present whether this is an ordinary "forward" operation or the undo operation of a CLR.
- **LSN** identifies this redo log record uniquely on its log. The LSN is used to control the redo scan and checkpointing of the log. It is not stored in either log records or in BLOCKs.

5.2 Checkpointing

The starting point in the log for the "redo scan" is called the "safe point". It is safe in two senses. First, redo recovery can safely ignore records that precede the safe point since those records are all already included in the versions of BLOCKs in PSTORE (or ASTORE). Second, these records might be truncated from the log because they are no longer needed. This later will be surely true for pure redo logging, and at times will be true for combined undo/redo logs, depending on the location of undo information for active transactions.

The purpose of checkpointing is first to ensure that the determination of the safe point, as described above, can survive system crashes. Secondly, this can be combined with a strategy for managing BLOCKs that permits the safe point to move so as to shrink the part of the log needed for redo. Checkpointing involves, among other activities, placing enough information on a log to enable enough of the state of the system to be reconstructed after a crash to assure recovery and to permit log management to continue.

A NODE's TLOG can be managed in isolation from other TLOGs of other NODEs. An MLOG is also managed in isolation from other MLOGs. There is a connection, however, between the management of a TLOG and its associated MLOG. Transaction recovery via the TLOG and media recovery via the MLOG will typically have different safe points that move at different times because MLOG checkpointing will be much less frequent than TLOG checkpointing. However, typically, a truncated part of an TLOG will continue to be required for media recovery. If so, the truncated part becomes part of the MLOG.

Some checkpointing schemes require that we know the safe point LSN for each BLOCK (called the RecLSN in [7]). This is used to re-calculate the TLOG safe point after the BLOCK with the oldest RecLSN is flushed to disk. The persistence of the RecLSNs across system crashes can be assured by including them in the checkpoint information. Further, to enforce the WAL protocol, we need to know the LSN of the log record whose action last updated each dirty BLOCK, which we call the LAST LSN. In many recovery schemes, this LAST LSN is stored in the BLOCK as the BLOCK LSN.

These LSNs are specific to a TLOG and hence describe the state of BLOCKs with respect to a particular TLOG. We observe that neither of these LSNs need be in the BLOCK itself. They can be managed separately from but in association with the BLOCK. This can be done via a Dirty BLOCKs Table that includes these entries for each dirty BLOCK entry. LAST LSNs needn't be stably stored as the last unlogged updates are lost in any event during a crash. Further, the BLOCK DSI of the version of each BLOCK in PSTORE substitutes for the BLOCK LSN in its role of controlling the application of redo operations to BLOCKs.

5.3 Normal Operation and Preparation for Recovery

During normal operation, steps must be taken with respect to logging so as to assure that recovery is possible. Our emphasis is on actions important to redo recovery.

A prominent characteristic of data sharing systems is the need for global cache management. Such cache management keeps track of which CACHEs contain which BLOCKs. This permits a NODE to request a BLOCK and for the most recent version of the BLOCK to be shipped to it. Cache management also keeps multiple NODEs from simultaneously updating a BLOCK. In Rdb/VMS [10], the VMS lock manager is used both for database locks and to support global cache management.

Because of the need for global cache management, we augment the list of operations that we describe to include not only the standard ones, but also the global cache management operations. The descriptions given for all operations emphasize the redo recovery aspects.

The actions are:

- Transaction Start:

  Write a START_TRANSACTION record to the TLOG.

- BLOCK Fetch:

  If the BLOCK is not currently held in the CACHE , notify the global cache manager. If the BLOCK is held by another node, the global cache manager sends a BLOCK request message to that NODE. The currently holding NODE eventually forces the BLOCK to PSTORE, marks the BLOCK as available, and notifies the global cache manager. The global cache manager then notifies the fetching NODE. This NODE usually will then read the BLOCK from PSTORE. This enforces the ONE-LOG redo requirement. Mark the newly read BLOCK as clean.

- BLOCK Update:

  Perform the concurrency control required so as to lock the BLOCK for update. In a data sharing system, this will involve use of distributed concurrency control. Perform a BLOCK Fetch if the BLOCK is not currently held in CACHE. Format redo information as a TLOG record and enter it in the TLOG buffer. Perform the action indicated upon

the version of the BLOCK in CACHE (perhaps using REDO to perform the action), updating the BLOCK's DSI with the ASI for the action. If the BLOCK was clean, make it dirty.

- BLOCK Return:

  Whenever BLOCKs leave a NODE's cache (a BLOCK Return), the global cache manager is notified. This may be either because a NODE needs the BLOCK's cache slot, or because a remote NODE has fetched the BLOCK. If the BLOCK is dirty, perform a BLOCK write to record the BLOCK in PSTORE and guarantee the ONE-LOG property. (BLOCK Writes are described below.) Then, inform the cache manager that the BLOCK is available. Finally, record that the BLOCK is not currently held.

- BLOCK Write:

  Since the BLOCK is dirty in the CACHE, enforce the WAL protocol by forcing to disk the TLOG records whose actions are recorded in the BLOCK. (Usually, no writing need be done because these records have already been written.) Then write the BLOCK to PSTORE. Finally, mark the BLOCK as clean in the CACHE.

- Transaction Abort:

  Apply the undo information for the transaction. Write a CLR in the TLOG for each undo action. The BLOCKs involved will usually already be locked. When this is not the case, locks should be acquired as if the undo operation were a regular "forward" BLOCK update. When this activity is complete, place a transaction termination[ABORT] record on the TLOG and release all locks held by the transaction.

- Transaction Prepare:

  (This is needed for two phase commit.) Write a PREPARE log record for the transaction to the TLOG, and force the TLOG up through this record. This force can be done as part of a group in which one force writes multiple prepare or commit records for multiple transactions.

- Transaction Commit:

  Write a COMMIT log record for the transaction to the TLOG, and force the TLOG up through this record. This can also be done as part of a group force. Release locks held by the transaction.

5.4 System Crash Recovery Processing

We describe system crash recovery in this section, with what has become its traditional three phases, analysis, redo, and undo.

5.4.1 Analysis Phase

The purpose of the analysis phase is to recreate relevant parts of the system state as of the time of the crash. Without an analysis phase, some extra work may be done during the other recovery phases. In [7], the information in the last complete checkpoint on the TLOG is read and used to determine which blocks were dirty at checkpoint time. TLOG records following this last checkpoint can then be read so as to bring this information up to date as of the time the system crashed.

The following is the kind of information useful for subsequent recovery.

- The redo safe point where the redo TLOG scan is to begin. TLOG records that precede the safe point have their effects already incorporated into the versions of BLOCKS that exist in PSTORE.
- The set of dirty blocks at the time of the crash. Only those blocks will need to have redo recovery. All other BLOCKS have all updates recorded in the versions present in PSTORE.
- The set of active or prepared transactions. Committed or prepared transactions must have their actions redone. Transactions active at the moment of crash will be aborted, and hence the effects of their actions must eventually be removed from the stable database. Only some of their actions may need to be redone (see [8]).
- The set of write locks held by active or prepared transactions. These locks need to be re-acquired prior to starting N-LOG undo. They are the locks that protect data changed by these transactions.

With node private TLOGs, the analysis phase can be done independently for each node. That is only the TLOG(s) of the node need be accessed.

5.4.2 The Redo Phase

All BLOCKs indicated as dirty in the analysis phase can be read into the CACHE. This read might be done in bulk, overlapped with the scanning of the TLOG(s). Some BLOCKs may be apparently dirty in several CACHEs and hence be read by several nodes to determine whether they need to be involved in local redo. Since a ONE-LOG version of every BLOCK exists in PSTORE, only one node can have records in its log that have a BSI equal to the DSI of the BLOCK. This node is the only one that will perform redo processing on the BLOCK. Hence, redo can be done in parallel by the separate NODEs of the system, each with its own TLOG. No concurrency control is needed during redo, even though some BLOCKs may be read by multiple NODEs.

The redo phase re-constructs the dirty BLOCK part of the CACHE state. The resulting CACHE contains the dirty BLOCKs in their states as of the time of the crash. The redo scan of the TLOG starts at the redo safe point. Hence, all updates to every BLOCK since it was written to PSTORE are assured of being included in the redo scan.

There are only two cases when trying to apply a TLOG record to its BLOCK.

1. The TLOG record’s BSI is not equal to the BLOCK’s DSI. The logged action can be ignored.
2. The TLOG record’s BSI is equal to the BLOCK’s DSI The appropriate "redo" activity is performed.

Our redo phase method involves repeating history, at least what [8] calls restricted repeating of history. Update redo log records, starting from the redo safe point are applied, sometimes even those that belong to "loser" transactions, i.e. those that will need to subsequently be undone. An action, to be redone, needs to be applied to the BLOCK in exactly the state to which the original action was applied. Sometimes this state was a state created by the action of a "loser" transaction, i.e. one that did not successfully commit.

In the application of a TLOG record to a BLOCK, the logged action is performed and the BLOCK DSI is updated to the ASI for the action. The global cache manager is notified of the presence of the BLOCK in this NODE's CACHE. At the end of the redo phase, the global cache manager knows the contents of all CACHEs.

5.4.3 The Undo Phase

We assume that undo recovery is N-LOG. Hence, the undo recovery phase needs concurrency control. Multiple nodes may need to undo the same BLOCK. However, note that normal database activity can resume once the redo phase ends, just as normal activity can proceed concurrently with transaction abort. All the appropriate locking is in place and the global cache manager knows the locations of all BLOCKs. This is ensured by not starting the undo phase until all NODEs have completed the redo phase.

We roll back all active transactions in the Active Transactions Table (but not prepared transactions). Undo processing can proceed in the same way as in rolling back an explicitly aborted transaction. In particular, for each undo action, a CLR is written to the TLOG that advances the state of the BLOCK to an SI that indicates that undo has been accomplished.

5.5 Media Checkpointing and Recovery

Media recovery has the same phases as system crash recovery. It differs in the following ways. What we present here is a high level description only.

- Rather than having each NODE process TLOGs individually and in parallel, a single NODE, called the MERGE NODE, is designated to perform the recovery.
- The MERGE NODE performs an analysis phase for each of MLOGs, so as to find the redo safe point for media failure. This safe point is the product of database backup, which is the MLOG analogy to TLOG checkpointing. We do not describe this backup process here. Possible schemes are described in [3,5,7].
- The MERGE NODE performs redo via our merge procedure so as to order the actions on the multiple logs appropriately.
- When media redo using the MLOGs is finished, TLOG recovery for the failed BLOCKs is performed, and this is done as with system crash recovery.

6 A New Distributed Database Organization

The capability provided by supporting multiple private logs in a data sharing system makes possible a new distributed database system model. Given the current and evolving hardware environment and the intrinsic physical limitations imposed by distance, this new model has some substantial advantages compared to currently popular ways of realizing distributed database systems.

6.1 Workstation Orientation

More important than improving the performance of data sharing systems in a cluster may be that private logs facilitate workstation-oriented client-server based architectures[4]. The workstation DBMS may manage some local data itself, and can share data with host computers and perhaps other workstations. The shared data can be cached ("checked out") for long periods and over multiple transactions in the workstation. Should another user, perhaps at another workstation, need access to some of this data, then the system writes the data back to the host, making this access possible. Only if there is heavy contention for data does this "Ping-Pong" effect adversely impact performance.

With private redo logging, no data need be written back to the host at transaction end, neither database pages nor log records. Private logs can be used to advantage by all the flavors of client-server architecture described in [4]. A private log at the workstation requires that the workstation have a PSTORE, e.g. a local magnetic disk. A workstation's PSTORE stably stores its private log. Further, this PSTORE might be used as a stable cache. This makes it possible for the workstation to perform checkpointing and log management activities without remote servers being involved, by making updated BLOCKs stable locally.

Having a system that supports both data sharing and partitioning permits each technique to be used where appropriate. In particular, data with high contention rates should be partitioned to avoid "Ping-Ponging" while data with low contention rates are good candidates for sharing. Indeed, it may even be possible to make the sharing/partitioning choice dynamically.

6.2 The Lending Library Paradigm

One can view a "mixed" sharing/partitioning system as a lending library. The hot or high contention data is treated like the books in the library's reference room. This is the partitioning part of the system where a server, i.e. the library, provides all the functionality. The cold or low contention data is treated like the ordinary books that are lent to users, i.e. the data is shipped to a user's workstation where it can be cached.

The identity and workstation location of cached data must be stably stored so as to persist across system crashes. This is analogous to a lending library recording who it is that has borrowed a book. Then, should the system fail, its recovery process can use this information so as to continue to protect the cached data from access. Importantly, the recovery process can then make the remaining data available once recovery is complete.

It is useful, in the above, to stably store the "distributed locks" used for distributed cache management. This information can be usefully exploited to permit shared as well as exclusive access to the data. The "lent" data that is shared can be cached by many workstations simultaneously. Such "locks" can be given up on request, of course, much as a borrower might return a book should he receive a call from the librarian that another patron has requested the book. This feature is already present in Rdb/VMS[10].

6.3 Coping with Workstation Failures

Perhaps the largest uncertainty involved with caching data at a remote workstation is how to make the cached data available again should the workstation crash or the network partition, and should this result in a very long delay. This is the "overdue book" problem for these systems and it can block further access to the failed workstation's cached data by other "lenders".

The analogy with lending libraries can be usefully exploited here. Borrowed books have "due dates" after which they are expected to be returned. For our "lending library" DBMS, workstations can have locks which time-out, with relatively long durations, and which must be refreshed [renewed] before their "due dates". At the due date, all locks on shared data can be broken while preserving transaction consistency. Workstations cannot be permitted to commit transactions that read this data after its "due dates".

Data cached for updating is more difficult to deal with. One approach might be to tolerate losing transactions whose effects are all still confined to the failed workstation. Such an approach would preserve the transaction consistency of the various server DBMS's, while sometimes requiring a workstation, when it recovers, to de-commit already committed transactions. To make this work requires that the workstation provide enough information to host systems for them to make the determination as to what data can be made available.

As with 2PC blocking, which is a more frequent problem for partitioned than for data sharing systems, a final recourse for "overdue data" might be to use "heuristic" methods. At this point, of course, the consistency of the database is no longer in the hands of the system. However, such methods, while extreme, can be successful if used with caution.

6.4 The Advantages of the Lending Library Paradigm

A "lending library" DBMS is sufficiently attractive as to more than compensate for the difficulty of coping with workstation failure. Its advantages include:

- The host DBMS is off-loaded, hence permitting it to support more users or reducing its cost. With private logs, this off-loading includes at least part of the cost of logging and of committing transactions. Since server processing power is more expensive than workstation processing power, a lower cost system can result. This is likely to be true in the future as well, as a workstation does not usually need the kind of powerful I/O subsystem which drives up the cost of servers.
- The workstation executes more of the database functionality. Workstation processing power is not only cheaper, it is less highly utilized than server processing power. This too is likely to continue as servers are usually configured for high utilization. Workstations are intended for high availability and hence tend to have low utilization.
- User response time can be reduced as the workstation is directly "in touch" with the user. Executing a transaction at the desktop avoids much of the message overhead and transmission delays associated with execution at a remote server. This is particularly important for interactive use and with object-oriented databases.

- System availability to execute distributed transactions will be higher. The data accessed by these transactions can be gathered from remote servers over an extended period of time and cached in a workstation. The workstation can then execute and commit this "distributed" transaction. All server DBMSs that have data in the transaction need not be available simultaneously. Potential time-outs that might arise during a two phase commit are also avoided, increasing the likelihood that transactions can commit successfully.

Private logs facilitate the first three of the above benefits. Private logs enable the last benefit. Hence, private logs enhance our ability to exploit the principal means that we have for improving the performance, availability, and responsiveness of distributed systems, namely local execution of functionality on data that is cached locally.

7 Discussion

We have presented a method that, by the use of multiple private redo logs, makes logging as efficient for data sharing systems as for partitioned or centralized systems. We have shown how those logs can be used in parallel for system recovery, and merged so as to provide media recovery. This merging is done without any notion of global timestamps and with no need for time synchronization between NODEs.

Our technique of storing before state identifiers in log records is a very general technique for supporting EXACTLY ONCE operation semantics. It is a form of the "testable state" approach. Because the state is testable, i.e. one can read the BLOCK's DSI, redo can be idempotent and hence be performed multiple times, even though the operations described in the redo log records are not idempotent. The BSI in the log record will match the DSI stored in a BLOCK exactly when the logged action needs to be redone, and only that state will result in the logged action being redone. This makes it possible to repeatedly call for the execution of an operation via messages whose delivery is uncertain and across multiple system failures. Each operation is guaranteed to be performed once and only once.

Variants of this technique might be used in a wide range of situations. In ATMs, for example, one can record the sequence number associated with the cash dispensing device in a "log record". This sequence number is mechanically incremented every time cash is dispensed, just as a BLOCK's DSI is incremented every time the BLOCK is updated. If the ATM crashes, it is still possible to assure that cash is dispensed exactly once by "redoing" the cash dispensing only when the ATM's current sequence number [DSI] still matches its sequence number [BSI] as recorded in the log record.

The closing of the recovery performance gap between data sharing and partitioned or centralized systems makes it possible to seriously consider data sharing systems' significant advantages. These traditionally have involved fault tolerance, higher performance for certain applications, e.g. perhaps Debit/Credit, and easily scalable performance.

There may, however, be an even bigger advantage to node private logging. It facilitates a new way of looking at and organizing distributed database systems. The lending library paradigm has robustness and performance, in particular for relatively cold data, that may be difficult for purely partitioned systems to match.

8 Acknowledgement

Independently of our work, Ashok Joshi, Ananth Raghavan, T. Rengarajan, and Peter Spiro jointly conceived of a variant of BSI logging and log merging. Feedback from them was useful in the evolution this paper. Particularly valuable were conversations with Peter Spiro. Phil Bernstein, Butler Lampson, and Betty Salzberg made useful comments on early drafts of the paper.

9 Bibliography

1. Bernstein, P., Hadzilacos, V., and Goodman, N., *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.

2. Bhide, A. An analysis of three transaction processing architectures. Proc. of 14th VLDB, Los Angeles, CA, (Aug. 1988) 339-350.

3. Crus, R, Data recovery in IBM Database2. IBM Sys.J 23,2 (1984) 178-188.

4. DeWitt, D., Futtersack, P., Maier, D., and Velez, F. A Study of Three Alternative Workstation-Server Architectures for Object-Oriented Database Systems. Computer Sciences Technical Report #936 (May 1990) Univ. of Wisconsin-Madison, Madison, WI.

5. Gray, J. Notes on data base operating systems. Research Report RJ2188 (Feb 1978), IBM Research Division, San Jose, CA.

6. Kronenberg, N., Levy, H., Strecker, W., and Merewood, R., The VAXcluster concept: an overview of a distributed system. Digital Technical Journal No. 5, (Sept 1987) 7-21.

7. Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., and Schwarz, P., ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using Write-Ahead logging. Research Report RJ6649 (Jan 1989) IBM Almaden Research Center, San Jose, CA and ACM Trans. Database Systs. (to appear)

8. Mohn, C. and Pirahesh, H. ARIES-RRH: Restricted Repeating of History in the ARIES Transaction Recovery Method. Research Report RJ7342 (Feb 1990) IBM Almaden Research Center, San Jose, CA.

9. Mohan, C., Narang, I., and Palmer, J., A case study of problems in migrating to distributed computing: data base recovery using multiple logs in the shared disks environment. Research Report RJ7343 (Mar 1990) IBM Almaden Research Center, San Jose, CA.

10. Rengarajan, T., Spiro, P., and Wright, W., High availability mechanisms of VAX DBMS software, Digital Technical Journal No. 8, (Feb. 1989), 88-98.

11. Shoens, K. Data sharing vs. partitioning for capacity and availability. IEEE Database Engineering 9,1 (Jan. 1986) 10-16.

12. Shoens, K., Narang, I., Obermark, R. Palmer, J., Silen, S., Traiger, I., and Treiber, K. The Amoeba project. Proc. of IEEE Spring Compcon (1985), 102-105.

13. Stonebraker, M. The case for shared nothing. IEEE Database Engineering 9,1 (Jan 1986) 4-9.

14. Strickland, J., Uhrowczik, P., and Watts, V. IMS/VS: an evolving system. IBM Systems J. 21,4 (1982) 490-510.

15. Tandem Database Group. NonStop SQL, A Distributed, High-Performance, High-Availability Implementation of SQL. Tandem Technical Report 87.4, Cupertino, CA (April 1987)