A Recovery Algorithm for A High-Performance Memory-Resident Database System

Tobin J. Lehman
Computer Sciences Department
IBM Almaden Research Center

Michael J. Carey
Computer Sciences Department
University of Wisconsin-Madison

Abstract

With memory prices dropping and memory sizes increasing accordingly, a number of researchers are addressing the problem of designing high-performance database systems for managing memory-resident data In this paper we address the recovery problem in the context of such a system We argue that existing database recovery schemes fall short of meeting the requirements of such a system, and we present a new recovery mechanism which is designed to overcome their shortcomings The proposed mechanism takes advantage of a few megabytes of reliable memory in order to organize recovery information on a per "object" basis As a result, it is able to amortize the cost of checkpoints over a controllable number of updates, and it is also able to separate post-crash recovery into two phases—high-speed recovery of data which is needed immediately by transactions, and background recovery of the remaining portions of the database A simple performance analysis is undertaken, and the results suggest our mechanism should perform well in a high-performance, memory-resident database environment

1 INTRODUCTION

Memory-resident database systems are an attractive alternative to disk-resident database systems when the database fits in main memory [1] A memory-resident database system can offer significant performance improvements over disk-resident database systems through the use of memory-based, instead of disk-based, algorithms [Ammann 85, DeWitt 84, DeWitt 85, Lehman 86a, Lehman 86b, Lehman 86c, Salem 86, Shapiro 86, Thompson 86] With the potential performance gain, however, comes an increased dependency on the database recovery mechanism

A memory-resident database system is more vulnerable to failures Power loss, chip burnout, hardware errors, or software errors (e g , an erroneous program causing a "runaway" CPU) can corrupt the primary copy of the database A stable backup copy of the database must be maintained in order to restore all or part of the memory-resident copy after a failure Simple battery-backup memory is affordable, but it is not necessarily reliable and it is not immune to software errors, so it is not a complete solution to the memory-resident database system recovery problem Large memories that are both stable and *reliable* are both expensive and potentially slow—it is difficult to say just how expensive or slow these types of memories are because they are not widely available Using current technology as a guide, we believe that in the near future stable and reliable memory will be available in sizes on the order of tens of megabytes and will have read/write performance two to four times slower than regular memory of the same technology—too slow and too small to use for the entire main memory, but large and fast enough to use as a stable buffer Using this stable reliable buffer, we attempt to design of a set of recovery algorithms that both function correctly for memory-resident database systems and perform sufficiently well to warrant being used in a high-performance database system environment

1.1 Lessons Learned From Disk-Based Recovery Methods

Memory-resident database systems have recovery needs similar to those of disk-resident database systems Indeed, almost any disk-oriented recovery algorithm

This research was partially supported by an IBM Fellowship, an IBM Faculty Development Award, and National Science Foundation Grant Number DCR-8402818

[1]It is usually the case that the *entire* database will not fit in the amount of memory available, but it may often be the case that a large amount of the frequently accessed data will fit For the present, we are concerned only with the memory-resident database as a self-contained unit

would *function* correctly in a memory-resident database system environment In any database system, the main purpose of the recovery mechanism is to restore the primary copy of the database to its most recent consistent state after a failure A disk-oriented system is similar to a memory-resident system in the handling of transaction commit, as both types of systems need to record the effects of the transaction on some stable storage device In this section we describe how some disk-oriented recovery designs handle the transaction commit procedure Since we do not have space in this paper to describe these algorithms completely, we must refer the reader to the literature for further details [Kohler 81, Haerder 83, Reuter 84]

When a transaction has updated some portion of the database and is ready to make its changes permanent, there are several ways to handle the commit procedure

1 The transaction could flush all of its updated pages to the database residing on disk, as in the TWIST algorithm [Reuter 84]

2 The transaction could flush all of its updated pages to a separate device, thereby allowing it to write all of the dirty pages in one action using chained I/O, as in the Database Cache algorithm [Elhardt 84]

3 The transaction could make shadow copies of updated pages to simplify post-crash recovery, while still maintaining a record-level log of updates, as in the System R recovery algorithm [Lorie 77, Gray 81]

4 The transaction could write a record-level log of the updates that it has performed (using the write-ahead-log protocol [Gray 78]), and periodically take *checkpoints* to refresh the database and keep the amount of log data small, as in the Lindsay *et al* algorithm [Lindsay 79]

In [Reuter 84], Reuter analyzed the four methods mentioned above and found that, for a disk-oriented system, method (4) [Lindsay 79] outperformed the rest, method (3) [Lorie 77] was also found to perform well when the page table was memory-resident These methods are designed to produce little processing overhead during normal transaction processing, however, some log processing is required for transaction UNDO or system restart This appears to be the best approach for a high-performance database system, as UNDO processing is typically done only for approximately 3 percent of all transactions [Gray 78], and restart is needed rarely in most systems The Database Cache and TWIST algorithms, on the other hand, involve larger processing overheads for normal transaction processing, as they are designed to provide support for fast UNDO processing and fast system restarts

From examining disk-oriented database systems, then, it appears that method (4) would provide a good basis for a memory-resident algorithm Its write-ahead log protocol using record-level log records appears to be a good method for handling the transaction commit process However, the problem with using such an "off the shelf" disk-oriented recovery scheme for a memory-resident database system is that the performance of the recovery mechanism would probably be poor in those cases where the entire memory-resident database, the "buffer", must be written to disk (as in a checkpoint operation) or read from disk (as in a system restart situation) Furthermore, a memory-resident database system appears to be able to obtain performance gains by eliminating the buffer manager altogether [Lehman 86a, Lehman 86c], so buffer-oriented recovery algorithms must be modified to reflect a "memory-resident" approach rather than a "buffer-pool" approach We must take a closer look at the requirements of a memory-resident system in order to design proper memory-resident database recovery algorithms We first examine previous work on memory-resident database system recovery [DeWitt 84, Ammann 85, Eich 86, Hagmann 86, Leland 85, Salem 86]

1.2 Memory-Resident Database Recovery Proposals

Recovery designs for memory-resident database systems have not been very different from those for disk-resident database systems, with one notable exception IBM's IMS FASTPATH [IBM 79, IBM 84] FASTPATH was the first to introduce the notion of *commit groups* The basic idea of commit groups is to amortize the cost of log I/O synchronization over several transactions rather than just a single transaction Where a single transaction would normally wait for its log information to be flushed to disk before committing and releasing locks, group commit allows it to *precommit*, whereby its log information is still in volatile memory (not yet flushed to disk), but its locks are released anyway The log information of several transactions accumulates, being written to disk when the log buffer fills up Finally, once the log information arrives safely on disk, the transaction officially commits This technique allows transactions accessing the same information to "overlap" somewhat, thus increasing concurrency and transaction throughput Note that there is no danger of a database update arriving at the disk before the corresponding log record, as the database update stays (only) in memory, in the special case of a checkpoint, the log would be forced to disk first DeWitt *et al* [DeWitt 84] point out that a stable log buffer memory can also be used to allow transactions to commit without log I/O synchronization, at the expense of making the log buffer memory both stable and reliable A stable log buffer provides the additional advantage of allowing the recovery mechanism to post-process the committed log data, performing log compression or change accumulation

In performing database checkpoints, the memory-resident database system recovery proposals do not differ much from disk-oriented methods, they flush the dirty portion of the buffer to stable storage DeWitt *et al* [DeWitt 84] propose first creating a shadow copy of the dirty portion of the database and then writing it to disk Eich [Eich 86] proposes writing the dirty portion of the database to disk when the database system naturally quiesces (though this seems likely to be a rare event in a high-performance database system) Finally, Hagmann [Hagmann 86] proposes simply streaming the entire memory copy of the database to disk for a checkpoint In a sense, these methods each treat the database as a single object instead of a collection of smaller objects—for post-crash recovery, these methods will reload the entire database and process the log before the database is ready for transaction processing to resume

It is often the case that a transaction can run with only a small portion of the database present in memory A more flexible recovery method would recover the data that transactions need in order to run on a demand basis, allowing transaction processing and general recovery to proceed in parallel We propose a design for a recovery component that provides high-speed logging, efficient checkpointing, and a post-crash recovery phase that enables transaction processing to resume quickly In the next section we describe our new memory-resident database system recovery algorithm In Section three, we provide a simple analysis that supports our claim of high performance Section four concludes the paper

2 A NEW RECOVERY METHOD

In order to describe our recovery scheme more clearly, we need to describe it in the context of our intended Main Memory Database Management System (MM-DBMS) architecture [Lehman 86b] The main feature of relevance here is its organization of memory Every database object (relation, index, or system data structure) is stored in its own logical segment Segments are composed of one or more fixed-size partitions, which are the unit of memory allocation for the underlying memory mapping hardware (We use the word partition rather than page to avoid any preconceived notions about the uses of a partition) Partitions represent a complete unit of storage, database entities (tuples or index components) are stored in partitions and do not cross partition boundaries [2] Partitions are also used as the unit of transfer to disk in checkpoint operations

2.1 Overview of Proposal

The proposed memory-resident database recovery scheme uses two independent processors, a main processor and a recovery processor, stable memory comprising two different log components, a Stable Log Buffer and a Stable Log Tail, and disk memory to hold both a checkpoint copy of the database and log information The two processors run independently and communicate through a buffer area in the Stable Log Buffer

The main CPU performs regular transaction processing–its only logging function is to write a transaction's log records to the Stable Log Buffer The recovery manager, running on the recovery CPU, reads log records from the Stable Log Buffer that belong to committed transactions and places them into bins (called *partition bins*) in the Stable Log Tail according to the address of the partition to which they refer Each partition having outstanding log information is represented in the Stable Log Tail by such a partition bin Each partition bin holds REDO log records and other miscellaneous log information pertaining to its partition Partitions having outstanding log information are referred to as *active*

As partition bins become full, they are written out to the log disk, log pages for a given partition are chained together for recovery purposes Grouping log records according to their corresponding partitions in the Stable Log Tail allows the recovery manager to keep track of the update activity on individual partitions When a partition has accumulated a specified threshold count of log records, it is marked to be checkpointed, and thus the cost of checkpoint operations are amortized over a controlled number of update operations Grouping log records also has another significant advantage—it allows partitions to be recovered independently

After a crash, the recovery manager restores the database system catalogs and then signals the transaction manager to begin processing As each transaction requests access to relations and indices, the transaction manager checks to make sure that these objects are available, and, if not, initiates recovery transactions to restore them on a per partition basis

In the remainder of this section the supporting hardware for the recovery mechanism is presented, then, details of the recovery algorithms, including regular logging, checkpointing, recovery, and archive logging, are described A simple performance analysis of the logging, checkpointing, and recovery algorithms is provided in Section 3

2.2 Hardware Organization

The hardware architecture for the recovery mechanism is composed of a recovery CPU, several megabytes of reliable, stable main memory, and a set of disks The recovery CPU has access to all of the stable memory, and the main CPU has access to at least part of the stable memory The two CPU's *could* both address all of memory (both volatile and nonvolatile), or they could share only the address space of the Stable Log Buffer To allow more flexibility in the actual hardware design, and also to provide better isolation of faults, our algorithms require the two CPU's to share only the address space of the Stable Log Buffer, using it as a communication buffer along with its other uses Also, though the recovery duties could be performed by a process running

[2]Long fields, such as those used to hold voice or image data, are managed by a separate mechanism not described here

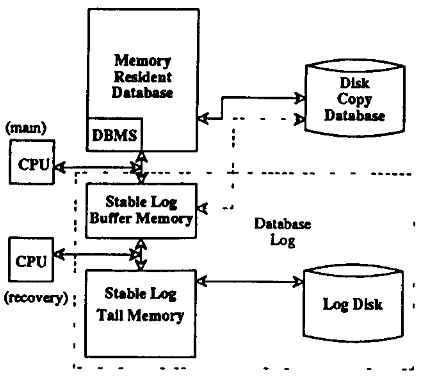

Figure 1 Recovery Mechanism Architecture

on one of many processors of a multiprocessor, a dedicated recovery processor (or even a dedicated recovery multiprocessor) can take advantage of specialization by having closer ties with the log disk controllers and running a special minimal operating system The disks are divided into two groups and used for different purposes One set of disks holds checkpoint information, and the other set of (duplexed) disks holds log information The recovery CPU manages the log disks, and both the recovery CPU and the main CPU manage the checkpoint disks Figure 1 shows the basic layout of the system

The two processors have logically different functions The main CPU is responsible for transaction processing, while the recovery CPU manages logging, checkpointing operations, and archive storage Although the two sets of tasks could be done by a single processor, it is apparent that there is a large amount of parallelism possible Indeed, although only two processors are mentioned, each CPU could even perhaps be a multiprocessor which further exploits parallelism within its own component In discussing the design, however, we refer to only two CPUs

2.3 Regular Logging

The logging component manages two logs one log holds regular audit trail data such as the contents of the message that initiates the transaction, time of day, user data, *etc* , and the other holds the REDO/UNDO information for the transaction The audit trail log is managed in a manner described by DeWitt *et al* [DeWitt 84] and uses stable memory We concentrate on the REDO/UNDO log, as it is responsible for maintaining database consistency and is the major focal point of recovery

The REDO/UNDO logging procedure is composed of three tasks First, transactions create both REDO and UNDO log records, the REDO log records are placed into the Stable Log Buffer, and the UNDO records are placed into a volatile UNDO space Second, the recovery manager (running on the recovery CPU) reads log records belonging to committed transactions from the Stable Log Buffer and places them into partition bins in the Stable Log Tail Last, the partition bin pages of REDO log records in the Stable Log Tail are written to disk when they become full These three steps are discussed in more detail below

2.3.1 Writing Log Records

Transactions write log records to two places REDO log records are placed in the Stable Log Buffer and UNDO log records are placed in the volatile UNDO space REDO log records are kept in stable memory so that transactions can commit instantly—they do not need to wait until the REDO log records are flushed to disk UNDO log records are *not* kept in stable memory because they are not needed after a transaction commits—the memory-resident database system does not allow modified, uncommitted data to be written to the stable disk database A log record corresponds to an entity in a partition a relation tuple or an index structure component An entity is referenced by its memory address (Segment Number, Partition Number, and Partition Offset)

Both the volatile UNDO space and the Stable Log Buffer (SLB) are managed as a set of fixed-size blocks These blocks are allocated to transactions on a demand basis, and a given block will be dedicated to a single transaction during its lifetime As a result, critical sections are used only for block allocation - they are not a part of the log writing process itself Because of these separate lists, transactions do not have to synchronize with each other to write to the log Therefore, having each transaction manage its own log record list greatly ameliorates the traditional "hot spot" problem of the log tail

The chains of log blocks for a transaction will appear on one of two lists, the committed transaction list or the uncommitted transaction list When a transaction commits, its REDO log block chain is removed from the uncommitted list and appended to the committed transaction list, and its UNDO log block chain is discarded (Figure 2) The committed transaction list is maintained in commit order so that the log records can be sent to disk in this order

2.3.2 Log Record Format

Log records have different formats depending on the type of database entity that they correspond to (relation tuples or index components) and on the type of operation that they represent All log records have four main parts

TAG	Bin Index	Tran Id	Operation

TAG refers to the type of log record, *Bin Index* is the index into the partition bin table where the log record will be relocated, *Tran Id* is the transaction identifier, and *Operation* identifies the REDO operation for the

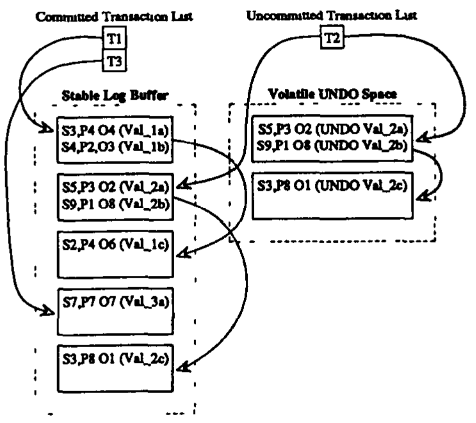

Figure 2 Writing Log Records

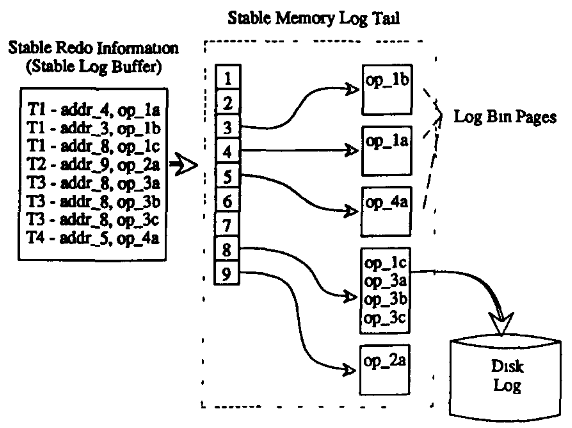

Figure 3 Regular Transaction Logging

entity The bin index is a direct index into the partition bin table, and it is used to locate the proper partition bin log page for an entity's log record Partitions maintain their partition bin index entries as part of their control information, so a bin index entry is easily located given the address of any of its entities (The next section discusses this in more detail)

Relation log records may specify REDO values for specific entities, so in one sense they are *value* log records However, they may also specify operations that entail updating the string space in a partition, which is managed as a heap and is not locked in a two-phase manner, so relation log records are really *operation* log records for a partition Index log records specify REDO operations for index components (e g , T Tree nodes or Modified Linear Hash nodes [Lehman 86c]) A single index update operation may affect several index components, so a log record must be written for each updated index component To maintain serializability and to simplify UNDO processing for transactions, index components and relation tuples are locked with two-phase locks [Eswaran 76] that are held until transaction commit

2.3.3 Grouping Log Records by Partition

The main purpose of the Stable Log Tail is to provide a stable storage environment where log records can be grouped according to their corresponding partitions The recovery manager uses this for several things

1 The log records corresponding to a partition are collected in page size units and written to disk The log pages for a partition are linked, thus allowing all the pages of a particular partition to be located easily during recovery

2 The number of log records for each partition is recorded, so the checkpoint mechanism can write those partitions that have a specified amount of log information to disk, thus amortizing the cost of a checkpoint operation over many update operations Once a partition has been checkpointed, its corresponding log information is no longer needed for memory recovery

3 Redundant address information may be stripped from the log records before they are written to disk, thereby condensing the log

Log records are read from the Stable Log Buffer and placed into partition bins in the Stable Log Tail (Figure 3) Each log record is read, its bin index field is used to calculate the memory address of the log page of the record's partition bin, and the log record is copied into that log page There are two main possibilities for organizing the log bin table there could be an entry in the table for every existing partition in the database, or there could be an entry for every active partition (Recall that an active partition is one that has been updated since its last checkpoint, so it has outstanding log information) Since the bin index should not be sparse, bin index numbers must be allocated and freed, like fixed blocks of memory If every partition were represented in the log bin table, then bin index numbers would be allocated and de-allocated infrequently—only as often as partitions are allocated and de-allocated However, if only the *active* partitions were represented in the log bin table, then bin index numbers would be allocated when partitions are activated and reclaimed when they are de-activated, thereby causing the bin index number resource manager to be activated more frequently For simplicity in design, we assume that each partition has a small permanent entry in the partition bin table This requires an information block in the Stable Log Tail for each partition in the database, but only active partitions require the much larger log page buffer The amount of stable reliable memory required for the Stable Log Tail depends on the total number of partitions in the database and the number of active partitions Each partition uses a small amount—on the order of 50 bytes, and each active partition requires a log page buffer—on the order of 2 to 16 kilobytes (depending on log page size)

Each partition has an information block in its log bin containing the following entries

- Partition Address (Segment Number, Partition Number)
- Update Count
- LSN of First Log Page
- Log Page Directory

The *Partition Address* is attached to each page of log records that is written to disk The entry serves as a consistency check during recovery so that the recovery manager can be assured of having the correct page It also allows the log pages of a partition to be located when the log is used for archive recovery

The *Update Count* and the *LSN of First Log Page* are monitors used to trigger a checkpoint operation for a partition The update count reflects the number of updates that have been performed on the partition When the update count exceeds a predetermined threshold, the partition is marked for a checkpoint operation The Log Sequence Number of the first log page of a partition shows when the partition's first log page was written, it is the address of the oldest of the partition's log pages When a partition is infrequently updated, it will have few log pages and they will be spread out over the entire log space The available log space remains constant, and it is reused over time The log space holding currently active log information is referred to as the *log window* The log window is a fixed amount of log disk space that moves forward through the total disk space as new log pages are written to it, so some active log information may fall off the end To allow log space to be reused, partitions are checkpointed if they have old log information that is about to fall off the end of the log window (There will actually be a grace period between when the checkpoint is triggered and when the log space really needs to be reused)

If the log space were infinite, all partition checkpoints would be triggered by the *update count* However, since the log space is finite, infrequently updated partitions will have to be checkpointed before they are able to accumulate a sufficient number of updates In this case, we say that those partitions were checkpointed because of *age* This works as follows The recovery manager maintains an ordered list of the first log pages of all active partitions Whenever the log window advances due to a log page being written, this *First LSN* list is checked for any partition whose first log page extends beyond the log window boundary When a partition becomes active it is placed on the First LSN list, and when it is checkpointed it is removed from the list The head of the list holds the oldest partition, so only a single test on this list is necessary when checking for the possibility of generating a checkpoint due to age

The *Log Page Directory* holds pointers to a number of log pages for a given partition (Figure 4) During recovery, REDO log records must be applied in the order that they were originally written If log pages were chained in order from most recently to least recently written, which is the reverse of the order *needed*, then log records could not begin to be applied until the last

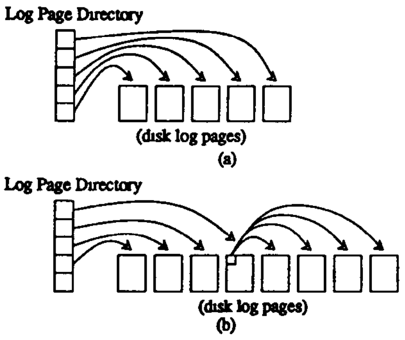

Figure 4 Log Page Directories

of the pages was read (which is the first page needed for recovery processing) Instead, a directory allows log pages to be read in the order that they are needed during recovery The size of the directory is chosen to be equal to the median number of log pages for an active partition so that, during recovery, it should often be the case that the log pages will be able to be read in order This allows the log records from one page to be used while the log records from the next page are being read off of the log disk

If fewer than N (the directory size) log pages have been written, the directory points to all of the log pages for the partition (Figure 4(a)) When more than N log pages have been written, the directory will be stored in every N^{th} log page

2.3.4 Flushing Log Records to the Log Disk

When the log records of a partition fill up a log page, the records are written to the log disk The recovery CPU issues a disk write request for that page and allocates another page to take its place (The memory holding the old page is then released after the disk write has successfully finished) The recovery CPU can issue a disk write request with little effort because it is a dedicated processor, it is using real memory, and it is probably running a single thread of execution (or at least a minimal operating system) It needs to do little more than append a disk write request to the disk device queue that points to the memory page to be written

2.4 Regular Checkpointing

As explained earlier, the main purpose of a checkpoint operation is to bound the log space used for partitions by writing to disk those partitions that have exceeded a predefined number of log records Its secondary purpose is to reclaim the log space of partitions that have to be checkpointed because of age When the recovery manager (running in the recovery CPU) determines that a partition should be checkpointed, either due to update count or age, it tells the main CPU that the partition is ready for a checkpoint via a communication buffer in the Stable Log Buffer The recovery manager enters the partition's address in the buffer along with a flag that represents the status of the checkpoint for that partition, initially this flag is in the *request* state, it changes to the *in-progress* state while the checkpoint is running, and it finally reaches the *finished* state after the checkpoint transaction commits A finished state entry is a signal to the recovery CPU to flush the remaining log information for the partition from the Stable Log Tail to the log disk

After a partition has been checkpointed, though its log information is no longer needed for memory recovery, the log information cannot be discarded because it is still needed in the archive log to recover from media failure If the partition has any log records remaining in the Stable Log Tail, they are flushed to the log disk In some situations, particularly when a partition is checkpointed because of age, a partial page of log records may need to be flushed to the log disk In that case, its log records are copied to a buffer where they are combined with other log records to create a full page of log information, thereby saving log space and disk transfer time by writing only full or mostly full pages to the log

Partition checkpoint images could be kept in well-known locations on the checkpoint disks, similar to a shadow page scheme, but that would require a disk seek to a partition's checkpoint image location for each checkpoint Instead, checkpoint images are simply written to the first available location on the checkpoint disks and a partition's checkpoint image location (in the relation catalog) is updated after each checkpoint Therefore, for performance reasons, the disks holding partition checkpoint images are organized in a pseudo-circular queue Frequently updated partitions will periodically get written to new checkpoint disk locations, but read-only or infrequently updated partitions may stay in one location for a long time (We use a pseudo-circular queue rather than a real circular queue so that partitions that are rarely checkpointed don't move and are skipped over as the head of the queue passes by) The checkpoint disk space should be large enough so that there will be sufficiently many free locations, or *holes*, available to hold new checkpoint images

A map of the disks' storage space is kept in the system catalogs to allow transactions to find the next available location for writing a partition New checkpoint copies of partitions *never* overwrite old copies Instead, the new checkpoint copy is written to a new location (the head of the queue), and installed atomically upon commit of the checkpoint transaction The relation catalog contains the disk locations of these checkpoint copies so that they can be located and used to recover partitions after a crash

The steps of the checkpoint procedure are as follows

1 The recovery CPU issues a checkpoint request containing a partition address and a status flag in the Stable Log Buffer

2 The transaction manager, running on the main CPU, checks the checkpoint request queue in the Stable Log Buffer between transactions For each partition checkpoint request that it finds, it starts a checkpoint transaction to read the specified partition from the database and write it to the checkpoint disk, and it also sets the checkpoint status flag to *in-progress*

3 The checkpoint transaction sets a read lock on the partition's relation and waits until it is granted Notice that a single read lock on a relation is sufficient to ensure that its relation and index partitions are all in a *transaction consistent* state, thus, only committed data is checkpointed

4 When the read lock on the partition's relation is granted, the checkpoint transaction allocates a block of memory large enough to hold the partition, copies the partition into that memory, and releases the read lock Relation locks are held just long enough to copy a partition at memory speeds, so checkpoint transactions will cause minimal interference with normal transactions The checkpoint transaction then locates a free area on the checkpoint disk to hold the partition (Since multiple checkpoint transactions may be executing in parallel, a write latch on the disk allocation map is required for this)

5 The updates to the disk allocation map and the partition's catalog entry are logged before the partition is actually written to disk Checkpoints of catalog partitions are done in a manner similar to regular partitions, except that their disk locations are duplicated in stable memory and in the disk log (because catalog information must be kept in a well-known place so that it can be found easily during recovery)

6 The partition is written to the checkpoint disk and the checkpoint transaction commits The memory buffer holding the checkpoint copy is released, the new disk location for the partition is installed in its catalog entry, and the status of the checkpoint operation is changed to *finished*

7 The recovery manager, on seeing the finished state of the checkpoint operation, flushes the partition's remaining log information from the Stable Log Buffer to the log disk

2.5 Post-Crash Memory Recovery

Since the primary copy of the database is memory-resident, the only way a transaction can run is if the information it needs is in main memory Restoring the memory copy of the database entails restoring the catalogs and their indices right away, then using the information in the catalogs to restore the rest of the database on a demand basis The information needed to restore the catalogs is a list of catalog partition addresses, and this is kept in a well-known location—it is stored twice, in the Stable Log Buffer and in the Stable Log Tail, and it is periodically written to the log disk Once the catalogs and their indices have been restored, regular transaction processing can begin

A Transaction could demand the recovery of a index or relation partition in one of two ways

1 It could predeclare the relations and indices that it required with knowledge gained from query compilation Then, when the relations and indices were restored in their entirety the transaction could run

2 It could simply reference the database during the course of regular processing and generate a restore process for those partitions that are not yet recovered Because of reasons having to do with holding latches over process switches (explained in [Lehman 87]), if a transaction made a reference to an unrecovered portion of the database while holding a latch, it would have to give up the latch or abort

There is a tradeoff—method (1) is simple, but it restricts the availability of the database by forcing the transaction to wait until the *entire* set of relations and indices that it requested are available before the transaction can run On the other hand, method (2) allows for more availability by restoring only those needed partitions, but it adds complexity and the possibility of several transaction aborts during restart It appears that experimentation on an actual implementation is required to resolve this issue

Using either method, transactions generate requests to have certain partitions restored The transaction manager checks the relation catalog for these entries to see if they are memory-resident If they are not, it initiates a set of recovery transactions to recover them, one per partition A relation catalog entry contains a list of partition descriptors that make up the relation, so the transaction manager knows which partitions need to be recovered Each descriptor gives the disk location of the partition along with its current status (memory-resident or disk-resident)

A recovery transaction for a partition reads the partition's checkpoint copy from the checkpoint disk and issues a request to the recovery CPU to read the partition's log records and place them in the Stable Log Buffer Once the partition and its log records are both available, the log records are applied to the partition to bring it up to its state preceding the crash (The processing of log records can be overlapped with the reading of log pages if the pages can be read in the correct order) Then, between regular transactions, a system transaction passes through the catalogs and issues recovery transactions (at a lower priority) for partitions that have not yet been recovered and that have not been requested by regular transactions

2.5.1 Reading in the Log Records

The operations specified by log records must be applied in the same order that they were originally performed A single backwards linked list of log pages would force the recovery manager to read *every* log page before it could even begin to apply the log records Recall that a *directory* of log pages is therefore used here, and since the directory size is chosen to be equal to the anticipated average number of log pages for a partition, it should be possible in many cases to schedule log page reads in the order that they were originally written Thus, the log records from one page can be applied to the partition while the next page of log records are being read When the number of log pages exceeds the directory size, it is possible to get to the first log page after $\lceil \frac{Number\ of\ Log\ Pages}{Number\ of\ Directory\ Entries} \rceil$ page reads

Relation log records represent operations to update a field, insert a tuple, or delete a tuple in a partition (More complicated issues involving changes to relation schema information are beyond the scope of this discussion) Index log records represent *partition-specific* operations on index components Recall that a single index update may involve several different actions to be applied to one or more index partitions For example, a tree update operation can modify several tree nodes, thus generating several different log records A given log record always affects exactly one partition

Letter	Represents
I	*Instructions*
N	*Number* (i e , a count)
S	*Size*
R	*Rate*
P	*Processing Power*

Table 1 Variable Conventions

2.6 Archive Logging

The disk copy of the database is basically the archive copy for the *primary* memory copy, but the disk copy also requires an archive copy (probably on tape or optical disk) in case of disk media failure Protecting the log disks and database checkpoint disks comes under the well-known area of traditional archive recovery, for which many algorithms are known [Haerder 83], so it is not discussed here

3 PERFORMANCE ANALYSIS OF THE RECOVERY PROPOSAL

To get an idea of how this recovery mechanism will perform, in this section we examine the performance of the three main operations of the recovery component logging, checkpointing, and post-crash recovery First, the logging capacity of the recovery mechanism is calculated to determine the maximum rate at which it can process log records Second, the frequency of

Name	Explanation	Value and Units
I_{record_lookup}	Read one log record and determine index of proper log bin	20 Instructions / Record
I_{copy_start}	Startup cost of copying a string of bytes	3 Instructions / Copy
I_{copy_add}	Additional cost per byte of copying a string of bytes	0 125 Instructions / Byte
I_{write_init}	Cost of initiating a disk write of a full log bin page	500 Instructions / Page Write
I_{page_alloc}	Cost of allocating a new log bin page and releasing the old one	100 Instructions / Page Write
I_{page_update}	Cost of updating the log bin page information	10 Instructions / Record
I_{page_check}	Cost of checking the existence of a log bin page	10 Instructions / Log Record
$I_{process\_LSN}$	Cost of maintaining the LSN count and checking for possible checkpoints	40 Instructions / Page Write
$I_{checkpoint}$	Cost of signaling the main CPU to start a checkpoint transaction	40 Instructions / Checkpoint
I_{record_sort}	Total cost of the record sorting process	(Calculated) Instructions / Record
I_{page_write}	Total cost of writing a page from the SLT to the log disk	(Calculated) Instructions / Page
S_{log_record}	Average size of a log record	24 Bytes / Record
S_{log_page}	Size of a log page	8 Kilobytes / Page
$S_{partition}$	Size of a partition	48 Kilobytes / Partition
N_{update}	The number of log records that a partition can accumulate before a checkpoint is triggered	1000 Log Records / Partition
N_{log_pages}	Average number of log pages for a partition	(Calculated) Log Pages / Partition
R_{bytes_logged}	Byte rate of the logging component	(Calculated) Bytes / Second
$R_{records\_logged}$	Record rate of the logging component	(Calculated) Log Records / Second
$R_{checkpoint}$	Frequency of checkpoints	(Calculated) Checkpoints / Second
$P_{recovery}$	MIPS power of the recovery CPU	1 0 Million Instructions / Second

Table 2 Parameter Descriptions

checkpoint transactions is calculated for various logging rates, and the overhead imposed by checkpointing transactions is calculated as a percentage of the total number of transactions that are running Finally, the process of post-crash recovery is outlined for an individual partition and performance issues are discussed, and then the performance advantage of partition-level recovery is demonstrated by comparing it with database-level recovery

3.1 Logging, Checkpointing, and Recovery Parameters

Before we begin the analysis, we introduce the parameters that will be used throughout this section Table 1 gives the conventions that we use for the names of the parameters, and Table 2 lists each parameter of interest along with its meaning, value (determined as described below), and units

The environment used to generate these figures is based on a midsized mainframe with a 5 MIP uniprocessor for the main CPU and a smaller 1 MIP uniprocessor for the recovery CPU [3] The stable reliable memory for the recovery CPU is composed of the faster mainframe memory chips, but it is four times slower due to the complexity of making it stable and reliable [4]

Instruction counts per operation are estimated (this recovery scheme has not yet been implemented) and overheads produced by procedure calls, process switching and operating system interaction have been somewhat accounted for by padding the instruction counts for the operations Complicated microcoded instructions such as the block move instruction are represented as multiple instructions A *generic* instruction executed on the recovery processor is assumed to execute in one microsecond and a memory reference is assumed to execute in about one microsecond The reader should keep in mind that the instruction count numbers appear smaller than normal system numbers The recovery component is highly specialized and requires only a minimal operating system, and it has sole control of the log disks when they are actively receiving log information (The recovery component releases control of a log disk when that disk is transferred to the archive component to role the contents of the disk onto tape)

The disk parameters in Table 2 are based on a two-head per surface high-performance disk drive It uses two read/write heads per surface, so it has relatively low seek times The transfer rate for a track of data is double the transfer rate for individual pages, par-

[3] The speed of the main CPU is not used in any of the calculations presented here, but we mention it to put the reader in the proper frame of mind

[4] The cost and performance figures of this stable memory are projected from current technology This memory is not available today, but we believe it will be widely available within the next decade

titions are written in whole tracks, whereas log pages are written individually (requiring separate disk operations) To achieve the maximum transfer rate possible for writing log pages, the disk sectors of the log disk are interleaved, logically adjacent sectors are physically one sector apart (For simplicity, sectors are assumed to be the size of one page) After writing one page, a disk needs a small amount of *think time* to set up for the next page write—more time, we assume, than the time it takes to travel from the end of one sector to the beginning of the next physically adjacent sector Therefore, by logically interleaving the sectors, the disk has the time of one full sector to reset for the next page write We also use different seek times for the checkpoint disks and the log disks The seek time for a partition read is an average seek time, as a partition can be anywhere on the disk in relationship to the disk head during the recovery process However, even though the log pages for a partition will be spread out over the log space, each page will be relatively close to its sibling, so the seek times between log pages should be somewhat less than the average seek time

Picking an optimal size for partitions and log pages involves dealing with a list of tradeoffs For example, the log page size represents a subtle tradeoff between the space required to hold log pages in the Stable Log Buffer and the frequency of page writes and page allocations The partition size affects several factors the number of entries in the Stable Log Tail, since larger partitions mean fewer partition entries, the cost and efficiency of checkpoints, since larger partitions might cause a larger percentage of non-updated data to be written during a checkpoint operation, and the overhead of managing partitions, since smaller partitions mean maintaining more entries per relation The sizes for log pages and partitions in Table 2 were chosen from the middle of a range of possible values, given the specifications and database reference patterns of a particular database system, it may be possible to pick better page size and partition size values

3.2 Logging Capacity Analysis

The logging rate of a recovery mechanism must be greater than the rate at which the main CPU can generate log records, or else the recovery mechanism will be the performance bottleneck of the system We estimate the logging capacity of the proposed design using a simple analytical model [5] During normal processing the recovery CPU spends most of its time moving log records from the Stable Log Buffer into partition bins in the Stable Log Tail, it spends a smaller portion of its time initiating disk write requests for full pages of log records and, an even smaller portion of its time is

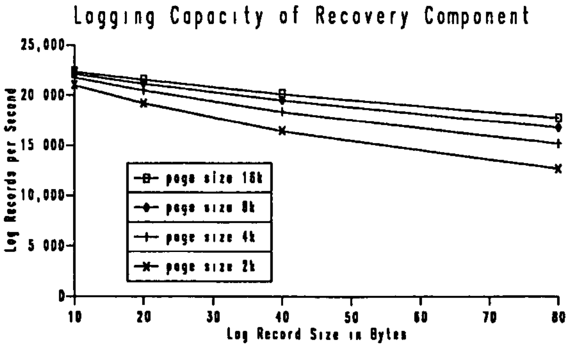

Figure 5 Graph 1—Logging Speed

spent notifying the main CPU of partitions that must be checkpointed

The cost of the first aspect of recovery processing is the cost of the *sorting* process of moving a single log record from the Stable Log Buffer to its correct location in the Stable Log Tail

$$I_{record\_sort} = I_{record\_lookup} + I_{page\_check} + I_{copy\_start} + (I_{copy\_add} * S_{log\_record}) + I_{page\_update} + I_{page\_alloc} * \frac{S_{log\_record}}{S_{log\_page}}$$

The second and third aspects of the logging process entail writing the partition bin log pages to disk as they fill up and notifying the main CPU each time a partition needs to be checkpointed

$$I_{page\_write} = I_{write\_init} + I_{process\_LSN} + \frac{I_{checkpoint}}{N_{update} * \frac{S_{log\_record}}{S_{log\_page}}}$$

The recovery processor executes a number of instructions to move a log record from the Stable Log Buffer to a partition bin in the Stable Log Tail I_{record_sort}, and it executes a number of instructions to write a partition bin log page to disk I_{page_write} If we combine those instruction costs in terms of instructions per byte and divide that into the processing power of the system (instructions per second) then we get the speed of the logging component in bytes per second

$$R_{bytes\_logged} = \frac{P_{recovery}}{\frac{I_{record\_sort}}{S_{log\_record}} + \frac{I_{page\_write}}{S_{log\_page}}}$$

Graph 1 shows the logging speed in log records per second (i.e., $\frac{R_{bytes\_logged}}{S_{log\_record}}$ for various log record and disk page sizes The number of log records generated by a transaction is of course application-dependent It can range from a few log records over hundreds of thousands of instructions (for computation-intensive transactions) to a few records over several thousand instructions (for Gray's debit/credit transactions [Gray 85]) to

[5] Note Space limitations force us to describe only briefly the underlying formulas on which the performance graphs are based For a more complete and narrative description of the formulas, the reader is referred to [Lehman 86b] The reader is also reminded that Table 2 summarizes the meaning of each parameter used in the analysis

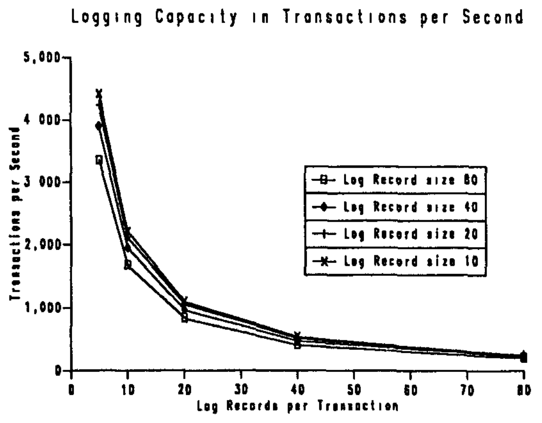

Figure 6 Graph 2—Transaction Rates

one log record over only hundreds of instructions (for update intensive transactions) Graph 2 shows the various maximum transaction rates that can be supported by the logging mechanism as the number of log records generated by a transaction is varied

Typical log records should be small, as common operations such as index operations, numerical field updates, and delete operations all generate log records that are 8 to 24 bytes in size Larger log records will be generated by other operations - updates to long fields or insertions of whole tuples, for example - but these are expected to occur less frequently Gray's notion of a typical debit/credit transaction is one that writes approximately four log records Given four log records per transaction, our logging component estimated capacity is approximately 4,000 transactions per second—a figure sufficiently high to suggest that the logging component will probably not be the bottleneck of the system [6]

3.3 Checkpointing Overhead Analysis

The recovery processor does little work for checkpointing When it notices that an object should be checkpointed, it simply signals the main CPU that the task should be done The major overhead lies with the main CPU, as it performs the real work of a checkpoint operation It is responsible for locking the object, copying it to a side buffer and then releasing the lock on the primary copy of the partition, locating a disk track for the partition, logging the updates to the disk map and catalog information, scheduling the disk write, and finally, committing the checkpoint transaction

The *frequency* of checkpoint transactions is of interest because it shows the percentage of all transactions that are devoted to regular transaction processing versus the percentage that are devoted to checkpointing partitions (Checkpoint transactions are relatively small, so they would be on the same order of computation as a debit/credit transaction, thus evaluating the cost of this checkpoint mechanism as a percentage of overall transaction load appears to be a valid measurement) The frequency of checkpoint transactions is determined by the logging rate, the update count for each partition, the number of active partitions, the distribution of updates over the active partitions, and the size of the log window (Recall that the log window is the active portion of the reusable log) For a given log window size, an active partition may reside in the log window long enough to accumulate enough updates to trigger a checkpoint, or it may not receive many updates, in which case it would be checkpointed because of age (so that its space may be reclaimed) The number of checkpoints triggered by update count depends on the number of active partitions for a particular log window and the distribution of log records over those partitions Given an infinite log window, checkpoints of all active partitions would eventually be triggered by update count, in which case the checkpoint rate would be:

$$R_{checkpoint} = \frac{R_{records\_logged}}{N_{update}}$$

This is the best possible scenario, as the cost of each checkpoint operation would be amortized over N_{update} update operations

Since log space is finite, there will be some active partitions that do not accumulate N_{update} updates before their log space needs to be reclaimed, so they will be checkpointed because of age instead This leads to a higher number of checkpoint operations, as the cost of checkpoints triggered by age are amortized over fewer than N_{update} update operations Thus, the worst case occurs when each active partition accumulates only one page of log records before it is checkpointed In this case [7]

$$R_{checkpoint} = R_{records\_logged} * \frac{S_{log\_record}}{S_{log\_page}}$$

It is not likely that the best or worst case will ever occur, instead, there will be some percentage of each type of checkpoint operation For a given number of active partitions, a large log window simply provides a better opportunity for a partition to accumulate N_{update} log records than a small window Thus, for performance reasons, there is a minimum log window size for a given number of active partitions—there should be at least enough pages in the low window to hold N_{update} log records for every active partition The only limitations on the maximum size are related to how much disk space is affordable

To calculate an average case checkpoint frequency, it is necessary to use a mix of checkpoints triggered by

[6]Note that we're not claiming that we can *produce* 4,000 transactions per second—we're simply claiming that the recovery component appears to be able to handle that rate of log records

[7]A partition does not take up disk log space until it has accumulated at least a page of log records

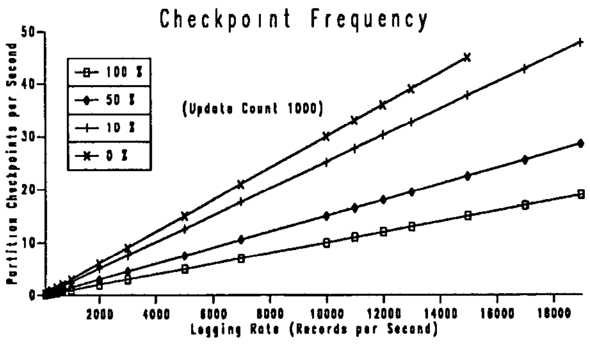

Figure 7 Graph 3—Possible Checkpoint Frequencies

age and triggered by update count For a given percentage of partitions that are checkpointed because of age, there are a range of possible checkpoint frequencies, as those partitions could have anywhere from a single page of updates to almost N_{update} updates For comparison purposes we will always assume the worst case—that a partition checkpointed because of age has accumulated only one page of log records Thus, the equation to determine the checkpoint rate for given fractions of partitions being checkpointed because of update count (f_{update_count}) and age (f_{age}) is

$$R_{checkpoint} = \frac{R_{records\_logged}}{f_{update\_count} * N_{update} + f_{age} * \frac{S_{log\_page}}{S_{log\_record}}}$$

Rather than try to choose actual numbers for the log window size, the number of active partitions, and the distribution of log records, we simply examine some different mixes of checkpoint trigger percentages Graph 3 shows the checkpoint frequencies for various update counts and trigger percentages as the logging rate is varied The log window size and the number of active partitions determine the number of checkpoints, but the logging rate determines how frequently they will occur—the logging rate determines how fast pages of log records go into the log window, and thus, how fast they reach the end of the log window, possibly triggering checkpoint operations For an update count of 1,000, if the log window size is large enough to allow 50 percent of the active partitions to accumulate N_{update} log records, then the checkpoint frequency will be fairly low Assuming an average transaction writes about 10 log records, this would indicate that the checkpoint transactions generated at this frequency would compose only 1 5 percent of the total transaction load An average number of log records less than 10 would simply decrease the percentage of checkpoint transactions

A larger update count causes fewer checkpoint operations to occur for a given trigger percentage, but it also increases the suggested minimum size of the log window A similar effect is seen when the log page size is doubled or the log record size is halved, as either one increases the number of log records that an active partition must accumulate before it can have a checkpoint triggered by age (Recall that an active partition's log information will not be written to the log disk until it has accumulated at least one full page of log records in the Stable Log Tail)

3.4 Discussion of Post-Crash Partition Recovery

The purpose of the partition-level recovery algorithm is to allow transactions to begin processing as soon as their data is restored Transactions issue requests for certain relations to be recovered (either "on demand" during the course of the transaction or predeclared at the beginning of the transaction) and they are able to proceed when their requested relations are available, they do not have to wait for the *entire* database to be restored An upper bound on the time needed to recover a relation is the sum of its partition recovery times A partition's recovery time is determined by the time it takes to read its checkpoint image from the checkpoint disk, to read all of its log pages, and to apply those log pages to its checkpoint image A partition's checkpoint image and its log pages may be read in parallel, since they are on different disks Also, provided that the log page directory was chosen to be large enough to hold entries for all of the log pages for the partition, the log pages can be read in the order that they were originally written, thus allowing the log records from one log page to be applied to the partition in parallel with the reading of other log pages (This assumes, of course, that the time required to apply a page of log records to a partition is less than the time it takes to read a log page) [8]

3.4.1 Comparison with Complete Reloading

The main alternative to partition-level recovery is database-level recovery An interesting point is that database-level recovery is a special case of partition-level recovery, with one very large partition (the entire database) An attractive feature of partition-level recovery is that it is flexible enough to perform pure partition-level recovery (read a partition, read its log information and recover it), full database-level recovery (read all partitions, read all of the log, recover all partitions), or some level in-between An optimization to pure partition-level recovery would be to load some significant portion of the log into memory during recovery so that the seek costs to read these pages would be

[8] Space considerations prohibit us from providing a more detailed description of the recovery costs We refer the interested reader to [Lehman 86b]

eliminated

4 CONCLUSION

Previous work in recovery, both for traditional disk-based database systems and for memory-resident systems, has addressed issues of logging, checkpointing and recovering a database Many designs have been proposed, but none of them have completely satisfied the needs of a high-performance, memory-resident database system Such a system needs a fast, efficient logging mechanism that can assimilate log records as fast or faster then they can be produced, an efficient checkpoint operation that can amortize the cost of a checkpoint over many updates to the database, and a post-crash recovery mechanism that is geared toward allowing transactions to run as quickly as possible after a crash

A design has been presented that meets these three criteria With the use of stable reliable memory and a recovery processor, the logging mechanism appears to be able to assimilate log records as fast as they can be produced Checkpoint operations for partitions are triggered when the partitions have received a significant number of updates, and thus the cost of each checkpoint operation is amortized over many updates Recovery of data in our design is oriented toward transaction response time After a crash, relations that are requested by transactions are recovered first so that these transactions can begin processing right away The remaining relations are recovered in the background on a low priority basis

5 Acknowledgements

Our thanks to Jim Gray, John Palmer, Jim Stamos and the SIGMOD referees for their helpful comments and suggestions on improving this paper

6 REFERENCES

[Amman 85] A Ammann, M Hanrahan, and R Krishnamurthy, "Design of a Memory Resident DBMS," *Proc IEEE COMPCON*, San Francisco, February 1985

[DeWitt 84] D DeWitt, et al, "Implementation Techniques for Main Memory Database Systems," *Proc ACM SIGMOD Conf*, June 1984

[DeWitt 85] D DeWitt and R Gerber, "Multiprocessor Hash-Based Join Algorithms," *Proc 11th Conf Very Large Data Bases*, Stockholm, Sweden, August 1985

[Eich 86] M Eich, *MMDB Recovery*, Southern Methodist Univ Dept of Computer Sciences Tech Rep [86-CSE-11, March 1986

[Elhardt 84] K Elhardt and R Bayer, "A Database Cache for High Performance and Fast Restart in Database Systems," *ACM Trans on Database Systems 9, 4*, December 1984

[Eswaran 76] K Eswaran, J Gray, R Lorie, and I Traiger, "The Notions of Consistency and Predicate Locks in a Database System," *Comm of the ACM 19, 11*, Nov 1976.

[Gray 78] J Gray, "Notes on Database Operating Systems," in *Operating Systems An Advanced Course*, Springer-Verlag, New York, 1978

[Gray 81] J Gray, et al, "The Recovery Manager of System R," *ACM Computing Surveys 13, 2*, June 1981

[Gray 85] J Gray, et al, "One Thousand Transactions Per Second," *Proc IEEE COMPCON*, San Francisco, February 1985

[Haerder 83] T Haerder and A Reuter, "Principles of Transaction-Oriented Database Recovery," *ACM Computing Surveys 15, 4*, December 1983

[Hagmann 86] R Hagmann, "A Crash Recovery Scheme for a Memory-Resident Database System," *IEEE Transactions on Computers C-35, 9*, September 1986

[IBM 79] *IBM IMS Version 1 Release 1 5 Fast Path Feature Description and Design Guide*, IBM World Trade Systems Centers (G320-5775), 1979

[IBM 84] *An IBM Guide to IMS/VS V1 R3 Data Entry Database (DEDB) Facility*, IBM International Systems Centers (GG24-1633-0), 1984

[Kohler 81] W Kohler, "A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems," *ACM Computing Surveys 13, 2*, June 1981

[Lehman 86a] T Lehman and M Carey, "Query Processing in Main Memory Database Management Systems," *Proc ACM SIGMOD Conf*, May 1986

[Lehman 86b] T Lehman, *Design and Performance Evaluation of a Main Memory Relational Database System*, Ph D Dissertation, University of Wisconsin-Madison, August 1986

[Lehman 86c] T Lehman and M Carey, "A Study of Index Structures for Main Memory Database Management Systems," *Proc 12th Conf Very Large Data Bases*, August 1986

[Lehman 87] T Lehman and M Carey, "Concurrency Control in Memory-Resident Database Systems," (submitted for publication)

[Leland 85] M Leland and W Roome, "The Silicon Database Machine," *Proc 4th Int Workshop on Database Machines*, Grand Bahama Island, March 1985

[Lindsay 79] B Lindsay, et al, *Notes on Distributed Databases*, IBM Research Report RJ 2571, San Jose, California, 1979

[**Lorie 77**] R Lorie, "Physical Integrity in a Large Segmented Database," *ACM Trans on Database Systems 2, 1*, March 1977

[**Reuter 80**] A Reuter, "A Fast Transaction-oriented Logging Scheme for UNDO Recovery," *IEEE Trans Software Eng SE-6*, July 1980

[**Reuter 84**] A Reuter, "Performance Analysis of Recovery Techniques," *ACM Trans on Database Systems 9, 4*, December 1984

[**Salem 86**] K Salem and H Garcia-Molina, *Crash Recovery Mechanisms for Main Storage Database Systems*, Princeton Univ Computer Science Dept Tech Rep CS-Tech Rep [034086, April 1986

[**Shapiro 86**] L Shapiro, "Join Processing in Database Systems with Large Main Memories," *ACM Trans on Database Systems*, September 1986

[**Thompson 86**] W Thompson, *Main Memory Database Algorithms for Multiprocessors*, Ph D Dissertation, Univ California-Davis, June 1986