LOG-DRIVEN BACKUPS: A RECOVERY SCHEME FOR LARGE MEMORY DATABASE SYSTEMS*

Eliezer Levy and Abraham Silberschatz

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

TR-89-24 September 1989

* This material is based in part upon work supported by the Texas Advanced Research Program under grant No. 4355.

Log-Driven Backups: A Recovery Scheme for Large Memory Database Systems*

Eliezer Levy
Abraham Silberschatz

Department of Computer Sciences,
University of Texas at Austin,
Austin, Texas 78712

September 18, 1989

Abstract

A new recovery scheme for main memory databases systems (MMDBS) is presented. The scheme is both practical and unique compared to other proposals in this area, since it is geared to accommodate databases that are not entirely memory-resident, and that use standard concurrency control mechanisms. It is assumed that a substantial portion of the database is memory-resident, and that the absent portion is accessed less frequently. Thus, the scheme capitalizes on the performance advantages offered by MMDBS, without precluding the possibility of having some portions of the database on secondary storage. The heart of the scheme is a novel and innovative approach to recovery processing in MMDBS, that eliminates expensive checkpointing activity. The advanced I/O technology of disk arrays is incorporated for the implementation of the approach.

*This material is based in part upon work supported by the Texas Advanced Research Program under grant No. 4355

1 Introduction

Spurred by the emerging technology of large and cost-effective semiconductor memory, researchers have devoted significant attention to the concept of main memory database systems (MMDBS). The potential for substantial performance improvement in MMDBS is promising, since I/O activity is kept at minimum. On the other hand, because of the volatility of main memory, the issue of failure recovery becomes more complex in this setting than in traditional, disk-resident database systems. Moreover, since recovery processing is the only component in MMDBS that must deal with I/O, this component must be designed with care so that it would not impede the overall performance. An overall perspective on the effects of large main memories on database systems can be found in [BI], and the references there.

In spite of the large size of current and future main memories (for instance, 1/2 Gigabyte of RAM memory is available in Princeton's GigaSun), it is always going to be the case that realistic database systems will not fit entirely in memory. Many of the proposed algorithms and schemes for MMDBS rely on the assumption that the entire database is memory-resident [GMS, LN, SGM2, DEW]. Although other proposals acknowledge that this assumption is not valid for practical reasons, the issue is not addressed directly in their designs [LC, HAG, EIC1]. We stress that the assumption that the database is only partially memory-resident *must* underly a practical design of MMDBS.

Another overlooked aspect in this area is concurrency control. Based on the claim that transactions never have to be suspended waiting for I/O in a MMDBS, it is said that transactions can run serially, one at a time [GMLV, LN]. Considering transactions that involve human interaction, or long-duration transactions in general, interleaved execution is indispensable. Therefore, the assumption of serial execution becomes actually a restriction.

In this paper we present a complete recovery scheme for MMDBS that is geared to accommodate databases that are not entirely memory-resident, and that incorporate standard concurrency control mechanisms. We do assume, however, that a substantial portion of the database is memory-resident, and that the absent portion is accessed less frequently. Thus, our approach capitalizes on the performance advantages offered by MMDBS, without precluding the possibility of having some portions of the database on secondary storage. The heart of the scheme is a novel and innovative approach to recovery processing in MMDBS. We propose to incorporate the advanced I/O technology of disk arrays for the implementation of our approach.

The paper is organized as follows. In Section 2 we introduce some terminology and concepts, establish our model of MMDBS, and then the recovery scheme is outlined. In Section 3, the role of disk arrays in our scheme is presented, and the basic feasibility of the proposed configuration is examined. Section 4 specifies the fundamental requirements from a design of our scheme. Sections 5 and 6 present in some detail alternatives designs, and motivate them. A complete algorithmic description of the scheme is given in Section 7, and a correctness proof is outlined in the following section. Our ideas are contrasted with related work in Section 9. We sum up with conclusions in Section 10.

2 The Framework

We first introduce some terminology, assumptions, and the technique of group commit. We then present the overall design of our recovery scheme and its hardware architecture.

2.1 Terminology and Assumptions

In the sequel we use the following terms to define our model. The *Primary Database* (PDB) is the part of the database that (currently) resides in main memory. On secondary memory (e.g., disk), there is an instance of the entire database, called the *Backup Database* (BDB). Data items, called *database records*, constitute the database.

The *log* of the database is a logical sequence of *log records*, whose prefix resides in stable storage. The log's suffix may be kept in a main memory buffer that is occasionally appended to the log on stable storage. We refer to the portion of the log in main memory as the *log's tail*, the buffer holding it as the *tail buffer*, and the rest of the log is referred to as the *stable log*.

We distinguish between sequential and random accessing of secondary storage for performance reasons. The stable log is accessed sequentially, whereas the BDB is accessed randomly.

The unit of transfer between main memory and secondary memory is a *page* (or a *block*). A page contains a fixed number of records. Log pages and database pages may be of different size.

We use the terms *Fetch*, and *Flush* to denote transferring an entire page from secondary memory into main memory, and vice versa, respectively. When we say that a record is *flushed safely* to a secondary storage medium, we mean that the entire page containing the record was written successfully.

Concurrency control is enforced by means of a *locking* protocol. We assume that the locking granularity is a single database record, and that the protocol produces *strict* schedules, e.g., Strict Two-Phase Locking [BHG].

We do not consider the issue of media failures in this paper. Also, we assume that flushing a page to the BDB is made atomic by stable storage techniques [LMP], or backup policies [SGM2].

2.2 Group Commit

Writing log records to the stable log one by one, as they are produced by the transactions incurs intolerable I/O overhead. An alternative, called *group commit* [DEW], is to buffer these log records in main memory, and flush them in groups, asynchronously from the transaction execution. Log records of executing transactions accumulate in a tail buffer that is flushed to the stable log once it is full. In this manner, the cost of I/O is amortized over a set of log records and a set of transactions.

The concept of *pre-commitment* [DEW] is usually used in conjunction with the group commit technique. A transaction release its locks as soon as its commit record is appended to the tail buffer (the pre-commit point), and before the record is actually written to the stable log (the commit point). Transactions, however, are not externalized (e.g., the user is not notified) until the actual commit point.

By releasing the locks before it commits, other transactions can read the pre-committed transaction's dirty data. We refer to these transactions as *dependent* transactions. Reading uncommitted data in this way does not lead to an inconsistent state as long as the pre-committed transaction actually commits before its dependent transactions. Observe that a pre-committed transaction is aborted only due to a system crash, and not because of a user or system induced abort (which we denote, hereafter, as an *intentional abort*). As long as log records are written to the stable log in the order they were appended to the tail buffer, a pre-committed transaction will have its commit record on stable log before the log records of its dependent transactions.

It has been observed [EIC2, SGM3] that group commit trades slower response time, for increased throughput. This is explained observing that transaction commitment is suspended until the buffer containing the commit record is full, and then a number of transactions commit simultaneously. An alternative method to speed up commit processing is by using non-volatile main memory. Maintaining the tail buffer in non-volatile memory, transaction commitment is made instantaneous. However, when non-volatile main memory is not available, there is no doubt that group commit is essential.

Recently, Li and Naughton propose a remedy for the response time problem [LN]. There, the timing of flushing the log buffer is controlled by another parameter, alternatively to having the buffer full. If the elapsed time since the first log record was appended to the buffer is too long, the buffer is flushed at once, even if it is not full. We use the group commit technique (in this improved form) in our scheme. The option of using non-volatile memory is discussed in Section 9.2.

2.3 An Overview of the Scheme

The scheme we propose combines group commit with a *log-driven backup* technique (defined shortly) and works as follows. After the tail buffer (whose size we assume, is one log page) is flushed safely to the stable log, it is not disposed of. Instead, it is made available for the *log processor*. This processor is a separate and dedicated processor, whose task is to update the BDB to reflect the modifications specified by the log records. While the log records in the old tail buffer are processed by this processor and updates to the BDB are issued accordingly, transaction processing is carried out independently, producing more log records that accumulate in the subsequent tail buffer.(essentially, double buffering of the tail buffer is employed).

Provided that the rate of producing log records and flushing them, is compatible with the rate the log processor consumes and processes there records, these two activities are synchronized and balanced. (We elaborate on this issue in Section 3). In effect, log records are pipelined in chunks of log pages from transaction processing, where they are produced, to the log manager, which records them on the stable log, and on to the log processor, where they guide the updates of the BDB (see Figure 1). Since the updates of the BDB are driven by the log records, we coin the name *log-driven backups* accordingly.

Independently from the above ongoing activity, database pages are exchanged between the PDB and the BDB, as dictated by the demands of the executing transactions. The *Paging Manager* is in charge of this exchange. We envision flushing of database pages to occur infrequently, and only in order to make room for newly fetched pages.

The actual updating of the PDB follows a *shadowing* technique (or *delayed updates* technique following the terminology of [SGM2]). Updates are recorded in a special, per transaction, buffer until the transaction's commit-processing phase. Updates are actually installed in the PDB only after a transaction is pre-committed. Thus, intentional aborts result simply in discarding the local buffers and does not incur any expensive Undo activity.

In order to avoid the overhead of copying shadows, one can use a technique dual to shadowing. In-place updates on the PDB are performed, and before images are maintained in buffers. In case a transaction commits, the before images are discarded, and in case the transaction aborts these images are used to restore the state of the updated records.

2.4 Hardware Architecture

The hardware architecture consists of a CPU that handles normal transaction processing (called the *main processor*), a separate CPU performing the task of the log processor, and a set of disks (see Figure 1). Apart from transaction processing, the main processor executes the paging manager and the log manager codes. Therefore, it might be beneficial to employ a multiprocessor for the main processor functions.

The main processor and the log processor need to share some address space that serves to store the tail buffers, and is used for other synchronization and communication purposes. The log processor can be a dedicated, less powerful processor that is specialized for its specific functions.

The main processor (actually, the log manager) manages a set of disks holding the stable log. Both the main processor (actually, the paging manager) and the log processor manage the BDB disks. That is, both processors can issue I/O requests to the controller of these disks.

3 Is the Log-Driven Backup Technique Feasible?

Before proceeding with the description of our scheme, we pause to assess an inherent difficulty with our approach, and show how to overcome it.

3.1 The I/O Problem

The observant reader must have noticed that the updates the log processor applies to the BDB are randomly scattered, and necessitate random accessing to the disk, whereas log records are appended in sequence to the end of the stable log. Therefore, updating the BDB is performed in a much slower rate, compared with the rate of producing log records and writing them to the stable log. As will become evident in this Section, the amount of BDB updates generated by the log-driven approach is quite formidable, and can pose problems to conventional disk architectures.

This problem is briefly mentioned in [LN]. The solution proposed there requires a second copy of the database to be maintained in main memory. Log records of committed transactions are applied to this second copy, and this copy is occasionally flushed to secondary storage in a long sequential write. This solution relies heavily on the assumption that the entire database resides in main memory. Moreover, it does not seem practical to have two copies of the database in main memory.

It seems that the only reasonable way to get around this I/O problem (without delaying transaction processing) is to increase the disk I/O bandwidth. In what follows we propose a solution, and evaluate its feasibility for a realistic, high-performance transaction processing system.

3.2 Using Disk Arrays to Overcome the I/O Problem

A well known technique of speeding disk data transfers is disk interleaving (or disk arrays) [KIM, PGK, SGM4, PB]. A single data unit is spread across a group of disks, instead of residing on a single disk. Ideally, if a data unit is spread across n disks, then it can be accessed in parallel, cutting the access time by a factor of $\frac{1}{n}$. Because of inherent fault tolerance problems that are due to the multiplicity of disks, this ideal speedup is actually unattainable.

In this form, disk interleaving is most attractive for supercomputing applications, where transfers of very large data blocks are dominant. Transaction processing systems, on the other hand, set different type of I/O requirements. A high rate of relatively small I/Os, in the order of couple of sectors, is needed. The use of disk interleaving for transaction processing systems is discussed in [PGK]. It is shown there how to configure an array of inexpensive disks (the type used for personal computers) in a way that provides both reliability and performance. The Level-5 Redundant Array of Inexpensive Disks (RAID) described there, enables performing many small I/Os in parallel. We briefly describe this type of RAID. Out of a group of n physical disks, $n - 1$ contain data, and a single one contains parity bits for that data. Entire transfer units are stored on the same disk (and are not interleaved) to enable performing small reads in parallel. Parity bits are computed for a set of bits on different disks, so that any single disk failure can be tolerated. In order to avoid contention, rather than having a dedicated parity disk, parity information is also interleaved across the disks. In short, each disk in the array is capable of performing small I/Os, independently. However, writes require up to four physical I/Os:

1. Read original data page
2. Read the associated parity
3. Write the updated data page
4. Write the updated parity

Since the parity is not a function of the written data page alone, reading the original data and parity is essential in order to compute the new parity. Using the analysis given in [PGK], this type of RAID incorporating D disks can provide over $D/4$ times the number of small writes of a single disk. An actual database system based on RAID is being currently built at Berkeley [SKPO].

We propose to store the BDB on the type of RAID just described. Modifying the BDB has to be done by means of flushing entire database pages. If records are big enough to constitute entire pages, then each log record processed by the log processor triggers an I/O. If this is not the case, the log processor can group records belonging to the same BDB page together, fetch the page, update it in memory, and then flush it back to the BDB. Fortunately, a page is read in any case before it is written as was explained above. Therefore, grouping updates on a page basis can reduce the amount of I/Os significantly. It is important to note that locality of reference might be very helpful in this context, since many updates to the same page can be grouped together.

Assuming that a transaction processing system runs at a rate of 1000 transaction per second and that an average, short duration transaction (e.g., Debit/Credit transaction [ANO]) produces about 5 updates, the system handles about 5000 database updates per second. (This update rate does not pose any problem in terms of log I/O, considering today's disks that feature a transfer rate of more than 3 Megabytes per second). An inexpensive disk, like the ones used for a RAID, is capable of performing up to 30 average I/Os per second, where an average I/O includes the average seek and average rotate times needed for a single sector access. Using over 600 disks for the mentioned RAID configuration, it is possible to handle the required amount of updates. (A similar, yet less conservative, analysis is carried out in [SKPO]). Observe that our analysis counted each and every update of a record as a separate write, ignoring the page-based grouping explained above.

Theoretically, such a RAID features huge usable storage capacity (up to 60 Gigabytes), and is quite cost effective. It should be noted however, that there is little if any, practical experience with RAID technology. Consequently, many issues still need to be addressed regarding the feasibility of this technology.

The above coarse analysis ignores the I/O traffic due to paging of database pages. However, in the context of paging I/O, we do note that the proposed RAID configuration performs well for large I/Os as well as for small I/Os. For large I/Os (called grouped I/Os in [PGK]), the multiple disks act as a single unit, each accessing a portion of a large data block in parallel. Here we have implicitly assumed that database pages exchanged between the BDB and the PDB are bigger granules than the pages the log processor writes to the BDB.

There are few more techniques that can reduce the amount of BDB I/O. First, the log processor should scan the log records in order to avoid overwriting updates. In case of updates that refer to the same record, only the most recent one should be actually written to the BDB. Scanning the log records, grouping them on a page basis, and eliminating overwrites, can be coupled with creating an optimal schedule of the disk writes to minimize seek time. In Section 5, we describe additional means of reducing the I/O to the BDB.

Employing these techniques will alleviate the I/O problem, but only when coupled with a radical approach like disk arrays, the problem can actually be solved. Other alternatives besides RAID exist. For instance, multiple-heads, parallel-access disks, where all tracks under a head can be accessed in parallel. In case access times of optical disks improve, another alternative might be arrays of optical disks. Because of the high reliability of this media, the design of arrays of optical disks might be simplified significantly. In any case, it seems that since the improvement in seek and rotation times is quite limited [DO, PGK], increasing the I/O bandwidth must be done via parallelism methods.

The goal of the above discussion was to illustrate the feasibility of the recovery scheme. However, all the ignored factors in the analysis must be carefully examined. Precise and thorough analysis of a particular scheme merits additional research.

4 Initial Design

We now make explicit several definitions and basic requirements implied by the intuitive description given so far.

We refer to the last log page that have been fully processed by the log processor, as the *safe page*. We define the sequence of log pages starting after the safe page and ending with the log page that has been completely written to the stable log, as the *stable tail* of the log (see Figure 2).

It was implicit in the informal overview of the scheme that the stable tail is always a single log page — the one currently processed by the log processor. Under these circumstances, the mentioned pipeline, through which log records pass, exhibits very strict synchronization pattern. The log manager and the log processor execute in lock-step mode — they are synchronized at the boundaries of each log page.

The components of the pipeline can be decoupled by introducing buffers bigger than one log page between them. We consider in detail the case where the stable tail may include more then a single log page. In this case, the log processor and the log manager, can operate asynchronously, each in its own pace. This asynchrony is an advantage considering that the rates of producing and consuming log records can vary drastically. The price paid, however, is the buffering capability needed to hold the stable tail pages. We do not wish to use the actual stable log as a buffer, since it implies violation of the sequential access mode of the log disk. Also, observe that the longer the stable tail is, the longer it takes to resume normal operation after a system crash (this intuitive statement is justified considering the precise algorithms given later). Hence, if a failure occurs when the stable tail happens to be rather long, resuming normal operation will be delayed accordingly.

Both the synchronous version and the asynchronous one can be characterized by the following general invariant:

I1. Numbering the ordered sequence of log pages starting with the safe page and ending with the the page in *the tail buffer, 0 to n, we assert the following:*

- *Page n is being filled up and occupies the tail buffer.*
- *Page $n-1$ is being flushed to the stable log by the log manager.*
- *Pages 1 through $n-2$ constitute the stable tail.*
- *A copy of Page 1 is being processed by the log processor.*
- *The updates of the safe page (page 0) have already been applied to the BDB.*

Figure 2 illustrate this invariant. In the synchronous case, n is constantly 3, whereas in the asynchronous case n ranges in a domain bounded by the buffering capabilities.

For the entire scheme to work correctly in the presence of arbitrary fetching and flushing of database pages from and to the BDB, the write ahead log protocol [GRY] must be followed. Its incarnation in this context is as follows:

R1. A dirty page is not flushed to the BDB before the corresponding log records are in the stable log. (The corresponding log records are all records representing updates to the particular page).

Since granularity of flushing to the log is log pages, *R1* implies that actually the complete log pages containing the mentioned update records must be in the stable log to allow a dirty database page to be forced to the BDB.

5 Specific Logging and Recovery Schemes

The general framework of our scheme has been set up. Details that still need to be specified are the timing and order of appending log records to the log, the contents of the log records, and the recovery procedure (to be used after a failure).

We devise three logging schemes, distinguished by the order in which log records of different transactions appear in log pages. Each scheme is appropriate for different kind of applications, and its advantages and disadvantages are outlined. In addition, we derive an efficient algorithm (described in Section 6) that coordinates the updating and paging aspects of accessing the BDB. Since the third scheme subsumes the first two, an explicit algorithm is given only for it (in Section 7).

Before we elaborate on the specific schemes we introduce more definitions and a lemma. The lemma dichotomizes the logging schemes.

We define a transaction as ***active*** from the time its ***begin transaction*** record appears on the stable log, until either a ***commit*** or ***abort*** record is written safely to the stable log.

We denote a value of a database record reflecting an update of some committed transaction as a ***committed value***. Accordingly, we define the ***last committed value*** of a record in some execution to be the last committed value written to that record in that execution. (The last two definitions follow [BHG]).

Lemma 1. The BDB contains only committed values regardless of the value of n in *I1* if, and only if, the stable tail always contains commit records for all transactions whose update records appear in it.

Proof: (if direction) The BDB can have an uncommitted record either as a result of an update performed by the log processor, or as a result of a page flushed from the PDB. The former case is impossible, since for it to occur, the commit record of the corresponding transaction should not be in the stable log at all, which contradicts the assumption. The latter case is impossible too, since a dirty page is flushed only after the corresponding log page becomes part of the stable tail (by *R1*). In which case, by assumption, the corresponding commit record is in the stable tail too, and hence there are no uncommitted values in the dirty page.

(only if direction) It suffices to consider the synchronous case ($n = 3$ in *I1*) to observe that this direction holds directly by the definitions of committed values and commit point.□

5.1 Restricted Logging Scheme L1

The first logging scheme we present is referred to as *L1*, and is characterized by the following constraint: All log records of a single transaction are contained within a single log page.

This constraint is severe and hence *L1* is appropriate only for the limited case of short transactions that make only few small updates. Appending log records to the tail buffer must be monitored in order to enforce the constraint.

Since in *L1* the log records of a transaction do not span more than one log page, Lemma 1 holds for *L1*. There are few immediate consequences. First, log records need not include before images, since no Undo processing is needed during recovery. This is justified by observing that uncommitted values in the PDB are lost in the event of system failure. Hence, if uncommitted values do not propagate to the BDB, no Undo's are needed during recovery. The last observation emphasizes the difference between traditional database systems and MMDBS.

The second consequence is that the recovery procedure for *L1* includes only a forward scan of the stable tail, during which all transactions appearing in the stable tail are redone.

5.2 Restricted Logging Scheme L2

Our second logging scheme is characterized by the following constraint: Log records of a single transaction appear contiguously in the log. We refer to it as *L2*.

Observe that the sequence of log records of a single transaction may span more than one log page. This sequence is contiguous, though. *L2* necessitates buffering the log records on a transaction basis, and appending them to the log tail in contiguous bulks. Consequently, all the logging activity has to be carried out after the transaction has pre-committed, at the commit processing phase. This might have a very undesirable effect, as pointed out in [EIC1], since transaction commitment is delayed by the log I/O.

In *L2*, as well as in *L1*, intentional aborts only incur discarding the local buffers of a transaction, since pre-commitment precedes any log I/O activity. Also, the identity of the transaction whose log records were being flushed to the stable log when the system failed can be deduced from the stable tail. This transaction is the only one that needs to be undone after a failure. On the other hand, *Lemma 1* does not hold for *L2*, and hence log records must contain before images in order to restore the BDB after a crash. Recapping the above observations, we note that the recovery procedure of *L2* undoes the updates of the single transaction mentioned previously. Undoing this transaction might cause a backward scan beyond the stable tail, but we are assured that once an update record of a different transaction is met, there are no more log records of the undone transaction. If the stable tail includes records of some committed transactions, only these updates must be redone (as was the case for *L1*).

5.3 General Logging Scheme L3

The last logging scheme, referred to as *L3*, is the most general one. The only constraint is the one implied by the group commit technique. More explicitly, the commit record of a pre-committed transaction is appended to the tail buffer before the commit records of any of its dependent transactions. As was mentioned in Section 2.2, it is easy to enforce this constraint.

Log pages may contain arbitrary mixture of log records, subject to the above constraint. Also, no buffering of the log records on a transaction basis is needed. Logging activity is done during transaction processing, and not only during the commit processing phase. It is important to notice that, in contrast to *L1* and *L2*, in *L3* log records are written to stable log before the pre-commit point. Obviously, Lemma 1 does not hold in such a case. Therefore, log records must contain before images as well as after images in *L3*. Also, at the completion of recovery processing of each log page, the log processor should record in a pre-fixed location on the BDB disk (or insert a log record containing) the list of currently active transactions. This list is necessary in order to identify and undo, at recovery time, aborted transactions that do not have any record in the stable tail, but have records in the stable log that might have triggered updates to the BDB. The log processor compiles this list according to the begin, commit, and abort records it tackles, while scanning the sequence of log pages. We call the record holding this information in stable storage the *recovery record*.

Another aspect unique to *L3* is the fact that intentional aborts can occur after the transaction has records in stable log, and updates installed in the BDB. As a result, careful Undo's need to be performed on the BDB following an intentional abort. There are three alternatives to handle these intentional aborts:

- *No Undo.* The log processor can delay applying log records of a transaction until a commit record is processed. In case that an abort record is tackled, the corresponding update records are discarded. This amounts to buffering log records on a transaction basis (as was done in *L2*) by the log processor. We reject this alternative since the amount of buffering can be too large for the simple log processor to handle. Also, since updates of eventually committed transaction are also delayed, recovery time is prolonged.

- *Undo from Log.* To undo the updates in the BDB, the corresponding before images can be found in the log. We reject this alternative too, since it requires random access to the log. Abandoning sequential access to the log implies significant decrease in throughput. As the crucial log I/O is preempted in favor of looking up before images, transactions are delayed unnecessarily.

- *Undo Records.* Since we use shadowing technique for updating the PDB, before images are available to an intentionally aborted transaction in its local buffers (avoiding the need to lock the PDB in order to get these images). It is possible to copy these before images into log records and add them to the log before the abort record. We call these type of log records, ***undo records***. If intentional aborts are not too frequent, this alternative is clearly preferred. The incurred overhead includes preparing the special undo records and their log I/O. If a system failure occurs before the undo records of an intentionally aborted transaction have been completely processed by the log processor, this transaction's effects should be undone similarly to a non-intentionally aborted transaction (see the complete recovery procedure below). We chose this alternative for our scheme.

A set of algorithms that summarizes scheme *L3* is presented in Section 7.

6 Coordinating the Updates to the BDB

The fact that the BDB is updated by both the log processor and by flushing pages from the PDB must be considered with care. First, one should wonder whether these double updates do not interfere with the correctness of this scheme. Second, since two identical updates are redundant, one of them should be avoided for performance reasons.

It turns out that the correctness aspect pertains only to *L3*. To see why this is the case, consider a record A containing the value v_0 and the following sequence of events:

1. Transaction T updates $A : A \leftarrow v_1$
2. The log record containing $< T, A \leftarrow v_1 >$ is appended to stable log
3. Transaction T is aborted
4. The undo record containing $< T, A \leftarrow v_0, Undo >$ is appended to stable log
5. The database page containing A is flushed to the BDB ($A = v_0$ in BDB)
6. The log processor processes the log record mentioned in (2) and sets $A \leftarrow v_1$ in BDB
7. A's page is fetched to PDB ($A = v_1$!!)

The above sequence is possible in terms of the temporal precedences described so far. However, the sequence results in the PDB being in an inconsistent state. The problem is exactly the doubled updates alluded to before. The log processor's update in step 6 overwrites a committed value with an uncommitted one, and the page is fetched to the PDB before the log processor is able to rectify the situation. This scenario cannot occur in *L1*, nor in *L2*, since in both there is no log I/O prior to the pre-commit point. The problem can be solved for *L3*, by imposing the following rule:

R2. The log processor does not apply updates to database pages that are <u>not</u> in the PDB. (This include updates in both update and undo records).

Implementing *R2* implies that the log processor should know which pages are in the PDB. In the algorithms given in Section 7, we assume that the log processor initially knows which pages are in the PDB, and it is notified about each page replacement by the paging manager. Thus, the exact synchronization between the log processor and the paging manager is specified. Alternatively, since a single controller serves I/O requests of both the log processor and the paging manager, enforcing *R2* can be implemented by a smart controller.

Besides the correctness aspect, *R2* enables the heavily loaded log processor to avoid processing some log records. *R2* dealt with cases where a page was flushed to the BDB before the corresponding updates were applied to the BDB by the log processor. I/O activity can be reduced considering the opposite case too, by imposing the following rule:

R3. When all of the log records corresponding to a page have been applied to the BDB by the log processor, flushing that page to the BDB is useless. The space it occupies in memory can be regarded free at once.

This rule can be easily implemented using Gray's [GRY] Log Sequence Numbers (LSNs). In our scheme, it suffices to increment the LSN each time a complete log page is flushed to the stable log. We associate an LSN variable with each page in the PDB. This variable should contain the greatest LSN of a log record concerning this page. Flushing of the page can be avoided if the page's LSN variable is at most the LSN of the safe page. In this case, we are assured that all the updates have been applied to the BDB already (by the log processor) and there is no need to flush the page. The LSN of the safe page must be available in shared memory, since it is updated by the los processor and read by the paging manager.

Implementing *R3* can be very effective in large memory systems, where we assume that paging activity is quite rare. By the time a page needs to be flushed to the BDB, it is quite possible that all the relevant updates have been propagated to the BDB by the log processor. We emphasize that incorporating *R3* is only for performance reasons, and has nothing to do with correctness.

7 Algorithms for L3

In this section we present a complete set of algorithms for implementing *L3*. The order of presentation is: transaction processing, log manager, log processor, recovery procedure, and the paging manager. The algorithms implement *I1* in its most general form. In Section 7.6 we specify the adjustments needed when considering the more synchronous version (i.e., where $n = 3$ in *I1*).

7.1 Transaction Processing

The following set of actions need to be considered with regard to the processing of a transaction.

read record r_i residing on page P_j

1. Fetch from the BDB page P_j, if it is absent from the PDB
2. Pin page P_j
3. Lock record r_i in shared mode
4. Read record r_i

Comment: We use the following semantics for the pin/unpin page operations:
A PDB page cannot be flushed to the BDB, unless it is unpinned by all transactions that have previously issued a pin request for it.

write record r_i residing on page P_j

1. Fetch from the BDB page P_j, if it is absent from the PDB
2. Pin page P_j
3. Lock record r_i in exclusive mode
4. Copy record r_i to a local buffer
5. Perform the update on the local buffer
6. Append the following update record to the tail buffer. If this is the first update of the transaction mark this record accordingly
   $<$ transaction ID, r_i, before image of r_i, after image of r_i $>$

7. Set $P_j.LSN \leftarrow tail-buffer.LSN$

Commit

1. Install the transaction updates in the PDB
2. Append the commit record to the tail buffer (*Pre-commit point*)
3. Release locks

Abort

1. For every database record updated by the transaction prepare the following Undo record using the local buffers and append it to the tail buffer: < transaction ID, record ID, before image, Undo >
2. Set the LSN's of the pages containing records updated by the transaction to the LSN of the tail buffer
3. Release locks, free the local buffers, and Unpin the pages pinned by this transaction
4. Append an abort record to the tail buffer < transaction ID, Abort >

Comment: An intentionally aborted transaction can be externalized as soon as it was decided to abort it.

7.2 Log Manager

The log manager is responsible for the log I/O.

1. When the tail buffer is full, or when a timing parameter has elapsed, flush the buffer to stable log
2. Signal the log processor that this log page is available for recovery processing
3. Allocate a new tail buffer with an incremented LSN
4. Externalize every transaction that had a commit record in the flushed page
5. Unpin pages that had been pinned by transactions that were committed in the flushed page$^{(A)}$ (implementing *R1*)

Comments:
(A) For *L2*, unpinning can be done for every page that had an *update* record in the flushed page. For *L3*, such early unpinning is possible, but does not make sense if the transaction eventually commits. This is because the page would be refetched in order to install the updates.

7.3 Log Processor

The log processor updates the BDB by grouping updates as was outlined in Section 3.2.

1. Get the first page in the stable tail and scan its records processing them as follows:
   - Update the active transaction list according to the begin, abort, and commit records
   - Eliminate overwriting update records
   - Prune all update records pertaining to pages not in the PDB (implementing *R2*)
   - Group the remaining update records according to the BDB pages
2. Arrange the pages to be updated optimally according to the disk scheduling policy

3. Fetch a page, update it, and flush it back to the BDB$^{(A)}$
4. Write in a pre-fixed location in the BDB's disk the following recovery record
   $<$ List of active transactions, *safe-page.LSN* $>^{(B)(C)}$
5. Write in a pre-fixed location in shared memory the *safe-page.LSN*

Comments:
(A) Fetching the page here, means simply reading it into memory. Flushing the page must be done carefully, as was mentioned in Section 2.1. Only in case that database records and the BDB pages are of the same size (as mentioned in Section 3.2), in-place updating can be used. This is because under such circumstances, a failure in the middle of a BDB update, causes only a single record to be lost, and its before or after image can be reconstructed from the log.
(B) This write must also be done carefully, using shadowing, and not in-place updating.
(C) Occasionally, the log processor may skip this action, and perform it only every couple of log pages. As a result, consecutive safe pages, will not be consecutive log pages. Also, only when the recovery record is written, does the stable tail actually shrink in size. Consequently, recovery may take longer if this action is omitted every once in a while.

7.4 Recovery Procedure

The general structure of this procedure is borrowed from [BHG]. It is invoked after a system crash.

1. Load a portion of the BDB into main memory and create the PDB
2. Create empty lists of committed and aborted transactions, denoted CL and AL respectively
3. Fetch the recovery record
4. Scan the stable tail backwards$^{(A)}$:
   If an undo record is read, add the transaction to AL
   If a commit (abort) record is read, add the transaction to CL (AL)
   If an update record of transaction T to record X is read then:
   (a) If T is in CL the record is ignored
   (b) If T is neither in CL nor in AL add T to AL
   (c) If T is (now) in AL then restore the before images of X (in both the BDB and PDB)$^{(B)}$

   If this update record is marked as the first one of T, remove T from AL
5. Once the backward scan exhausts the stable tail, add to AL all transactions in the active transactions list that are not in CL. Continue scanning backwards ignoring all records except update records of transactions in AL. These are processed as in (3.c) above. Terminate the backward scan once AL is empty$^{(C)}$
6. Scan the stable tail forward redoing updates of transactions in CL
7. Enable normal transaction processing

Comments:
(A) The location of the safe page, which marks one end of the stable tail, can be found in the recovery record. We assume that finding the last completely written log page, which marks the other end of the stable tail, is also possible.
(B) A more efficient recovery procedure would update only the PDB, and produce log records, so that the log processor would asynchronously update the BDB. Idempotence of such a procedure and its details deserve separate attention.
(C) The portion of the backward scan that goes beyond the stable tail can be made more efficient by chaining backwards records of the same transaction.

7.5 Paging Manager

The paging manager fetches database pages according to the demand of the transactions.

Fetch page P_i

1. Select from the pages that are not pinned a page, P_j, to be flushed$^{(A)}$
2. Notify the log processor about the change in the PDB constituency
3. If $P_j.LSN \leq$ *safe-page.LSN*$^{(B)}$
   then free the space in main memory (implementing *R3*)
   else flush P_j to BDB and free the space in main memory
4. Load P_i into main memory from the BDB

Comments:
(A) The selection can be based on the pages' LSNs, thereby making *R4* more effective.
(B) The LSN of the safe page is available in shared memory (see step 5 of the log processor).

7.6 Adjustments for the Synchronous Version

In order to suit better the case where the stable tail is a single log page, the above algorithms need to be modified as follows:

- The log processor and the log manger must synchronize after processing each log page.
- Since the stable tail is always a single page, there is no need to record the LSN of the safe page neither in the recovery record, nor in shared memory.
- Markers must be used to delimit log pages on the log, especially since partially-full log pages are appended to the stable log.
- After a crash, the recovery procedure must find the last marker, which identifies the end of the stable tail. As a safety precaution, the stable tail is assumed to consist of two pages. This is in order to avoid erroneous identification of the stable tail, when the last marker is written by the log manager before the log processor actually completes to process the next to last log page.

8 Proving the Scheme Correct

We are in a position now, to prove the correctness of our scheme. First, we consider our scheme when no system failures occur. When transactions are processed against a static set of pages comprising the PDB no correctness aspects arise. To guarantee correctness of the scheme in the presence of paging from the BDB, we must prove the following Theorem:

Theorem 1. A page fetched from BDB to the PDB contains the last committed value for each record in the page.
Proof: If a page is fetched from BDB it is absent from the PDB. That is, either it was never in the PDB, or it was previously flushed from the PDB. In the former case the claim holds vacuously, since no updates were applied to this page. Regarding the latter case, the idea is that when a page is flushed from the PDB

it contains all the last committed values, and by enforcing *R2* it is left intact onwards (until it is fetched again). Formally, by the semantics of unpinning, a page can be forced to the BDB, only after all transactions modifying its records have unpinned it. By the transaction processing algorithm, unpinning occurs only after the commit point of a transaction. In case of aborted transactions, since a shadowing mechanism is used for updating the PDB, the records' values are not affected at all. Therefore, a flushed page contains the last committed values for all its records. Now, by *R2*, this page is not modified any more (neither by update records nor by undo records) until it is fetched again. □
We note that in the case of scheme *L2*, where unpinning is triggered by the flushing of update records and not commit records, Theorem 1 must be weakened as follows: A page fetched from BDB to the PDB contains the last *pre-committed* value for each record in the page. Observe that this weaker form does not sacrifice correctness.

Theorem 2. When the recovery procedure completes, all database records have their last committed values.
Proof: Depending on the position in the log of the record defining the last committed value of an arbitrary database record, we consider the following cases:
Case 1. A database record whose last committed value appears as an after image in an update log record located prior to the stable tail. Such a record acquired its last committed value by either a log processor update, or when its enclosing page was flushed. This value is left intact by the recovery procedure regardless of whether the corresponding commit record is in the stable tail or precedes it.
Case 2. A database record whose last committed value appears as an after image in an update log record located in the stable tail. This value is restored by the forward scan of the recovery procedure (step 6).
Case 3. A database record whose last committed value appears as a before image in an update log record. Here, we must distinguish among the following subcases:
Subcase 3.1 Either an update record, or an undo record, or the abort record of the corresponding transaction appears in the stable tail. The transaction is added to *AL* in step 3.b, and therefore, all its updates are undone (by step 3.c).
Subcase 3.2 The abort record (and hence all update and undo records) of the corresponding transaction is located before the stable tail. In this case, the last committed value was restored by the log processor, when it processed the relevant undo record.
Subcase 3.3 There is no log record of the corresponding transaction in the stable tail, and its abort record is not in the log at all. Thus, the transaction was active, by definition, when the system crashed. Since there is no log record of this transaction in the stable tail, it must have been active when the log processor recorded the active transaction list corresponding to the safe page. Therefore, this transaction is added to *AL* in step 5, and all its updates are undone. □

9 Evaluation and Comparison with Related Work

As was mentioned in the Introduction, our scheme stands out by assuming that parts of the database may be non-memory-resident, and by accommodating standard concurrency control policies. In addition, it is distinguished by the use of the log-driven backup technique. In this section we compare our scheme with other alternatives, in light of this distinction. Also, we discuss the use of stable memory in the context of MMDBS recovery.

9.1 Comparison with Checkpointing Algorithms

Similarly to traditional disk resident databases, *checkpointing* is an activity that updates the BDB during normal operation in order to make recovery after a failure faster. There are several variations to MMDBS checkpointing (e.g., [DEW, PU, HAG]). A thorough survey of different checkpointing policies, their impact on overall recovery issues, and their performance can be found in [SGM1, SGM2].

In principle, all the cited algorithms make some sort of a sweep through the PDB and update the BDB accordingly. In the log-driven technique, the idea is to eliminate checkpointing altogether, since it impedes transaction processing. The BDB is kept up-to-date by applying updates as specified in the log records. It is the stream of log records that is used to guide a continuous update of the BDB, rather then a periodical sweep of the PDB.

Since we are going to refer to some specific checkpointing schemes later in this section, we briefly mention their names and key properties, as described in [SGM2]. Checkpointers can be classified either as *consistent* or *fuzzy*, according to the checkpoint they produce. A transaction-consistent checkpoint reflects a database state obtained when there are no transactions in progress. Producing a consistent checkpoint involves synchronization with executing transactions. Fuzzy checkpoints, on the other hand, may not reflect an atomic view of database operations. That is, the checkpoint may contain effects of incomplete transactions and other composite operations. The advantage of fuzzy checkpointing algorithms is that they require little, if any, synchronization with transaction processing. In principle, fuzzy algorithms amount to flushing of dirty pages, regardless of locks and other transaction activity. A fuzzy checkpointing algorithm is presented and analyzed in [HAG].

An algorithm, referred to as two-color algorithm, that produces transaction-consistent checkpoints is given in [PU]. This algorithm may cause transactions to abort in order to obtain the consistent checkpoint. Another consistent checkpoint algorithm, referred to as copy-on-update algorithm, is presented in [DEW]. There, transaction processing must be temporarily quiesced each time a checkpoint begins. Also, additional main memory space is required, since even after updates are installed in the PDB, old images must be preserved for the sake of the consistency of the checkpoint.

We list some points that contrast our algorithm with the above mentioned checkpointing algorithms:

- Checkpointing interferes in one way or another with transaction processing, since both activities compete for the PDB. In the extreme case, transactions have to be aborted (two-color algorithm). Even in fuzzy algorithms memory contention is inevitable since both normal transactions and the checkpointer must access the very same memory. By contrast, in the log-driven technique there is a clean separation between recovery processing and normal transaction processing. Therefore, recovery processing does not hinder the performance of transaction processing. Considering our scheme as a whole, we note that only the absolutely essential locking and logging mechanisms contribute to a delay in transaction processing.

- The separation between recovery-related activities and normal transaction processing is functional in nature, and hence can contribute to a better design. That is, the functions of transaction processing and recovery processing can be decoupled and performed by different processes (or different processors, as we have suggested). In general, this decoupling is applicable in systems employing checkpointing as well. However, the synchronization between the checkpointer the and transaction processor is tighter, and is characterized by the use of shared memory, whereas in the log-driven technique the synchronization has more of a pipeline nature.

- It has been observed in [SGM2] that consistent checkpoints must be supported by two copies of the database on secondary storage, since there is no guarantee that the entire checkpoint will be atomic. More precisely, there is always one consistent checkpoint of the entire database on secondary storage that was created by the penultimate checkpoint run, while the current run creates a new checkpoint.

  In contrast to consistent checkpoint algorithms, and similarly to fuzzy checkpointing, the log-driven approach need not be supported by two copies of the database on secondary storage. Care must be taken in order to make flushing a page atomic, but a single database copy suffices for this. (Essentially, a monoplex backup policy is sufficient [SGM2]).

- It is not clear how checkpointing algorithms can be adjusted to support our assumption of a partially resident database. The correctness of these algorithms may be jeopardized by arbitrary fetching and flushing of database pages. We conjecture that it is actually impossible to adopt a consistent algorithm for nearly-resident databases, since neither of the two copies maintained on secondary storage by these algorithms is an up-to-date BDB. It seems that fuzzy checkpointing, which is the simplest type of regular checkpointing algorithms, can be adopted for such purposes (actually it is used in traditional, disk-based databases), but this deserves separate attention.

The above comparison favors the log-driven approach. Among the rest, fuzzy algorithms seem to be close competitors. We note that fuzzy algorithms stand out (considering CPU overhead during normal operation) according to the performance evaluation studies of Salem and Garcia-Molina [SGM2]. The disadvantages of these algorithms are pointed out there too.

We now turn to evaluate the demerits of log-driven checkpointing:

- The log driven approach is constrained by the linear nature of the log. Each and every log record is processed and affects the BDB. Using regular checkpoints, records in the PDB may be overwritten in the course of the checkpointing, and hence not every update causes an update to the BDB. Consequently, assuming high degree of locality, regular checkpointing seems to be less sensitive to transaction load than the log-driven approach. One possible solution to this problem is to increase the buffering capabilities between the log manager and the log processor. During peak load periods, the length of the stable tail will increase, and during subsequent light load periods, the log processor will catch up and the stable tail will shrink again.

- A related problem of the log-driven approach is intentional aborts. Because all log records are processed, it is possible for an update of an intentionally aborted transaction to affect the BDB (as is the case in *L3*). Consequently, undoing such a transaction is made more complex and involves accessing the BDB (the undo records). The same problem exists for fuzzy algorithms. The only solution we envision for this inherent problem is in the spirit of *L1* and *L2* and the use of stable memory.

We should note that other methods that are log-driven in spirit can be found in [EIC1] and [LN]. However, both these methods have limited scope. In [EIC1], log records of a transaction are marked after the transaction has committed, so that only log records of committed transactions would affect the BDB. This, in turn, necessitates random accessing to the stable log. The deficiencies of the proposal of [LN] were already discussed in Section 3.

9.2 On the Use of Stable Memory

The literature contains numerous proposals focusing on non-volatile RAM memory (in short, stable memory) for MMDBS (e.g., [EIC2, SGM3, DEW, LC]). Stable memory has significant impact on recovery aspects, especially on the commit procedure. The basic guideline in traditional commit procedures is to spread the logging activity over the duration of the transaction. Thus, when the transaction is completed it can be committed instantly, as it does not need to wait for all its records to be flushed to the stable log. This guideline is exemplified in *L3* ,and in the group commit technique in general. Stable memory renders this guideline obsolete. Using stable memory for the tail buffer [EIC2, LC, DEW], transactions can commit as soon as their records are in the stable memory, regardless of whether the records have been flushed to disk or not.

A unique approach to MMDBS recovery that uses stable memory (mainly) for the tail buffer is presented in [LC]. We refer to this approach as the page-based algorithm. Stable memory is used extensively to achieve two desired effects. First, the expensive action of checkpointing is performed in a very disciplined manner. Instead of a periodic sweep of the PDB, checkpoints are applied to individual database pages each time these pages are updated more than a threshold number of times. In this manner, the cost of checkpoints is amortized over a controllable number of updates. Secondly, actual recovery is extremely fast, since there is no phase of loading the PDB to memory and applying the log records in order to reconstruct the entire database. Instead, the log is organized to enable recovery of individual data objects upon demand of executing transactions.

The individual page checkpointing is carried out as an ordinary transaction that locks the page before dumping it to the BDB. Thus, again regular transactions are delayed because of lock contention and since the same processor is responsible for both transaction processing and the checkpoints. This phenomenon, common to checkpointing algorithms in general, has no parallel in the log-driven technique.

Using non-volatile memory for the tail buffer may benefit our scheme too. The tail buffer can be made available for the log processor immediately, even before it is flushed to the stable log. This implies that the log processor may have to either operate at a higher rate, or buffer log pages during peak load periods.

Considering the actual logging scheme, the most suitable one in this context is *L2*. Recall that in *L2* log records are buffered on a transaction basis. We mention two advantages of using *L2* here. Separate buffering of log records on a transaction basis results in independent writing of log records by different transactions as was observed in [LC]. Transactions need not synchronize with each other in order to write the log. Only the buffer allocation is handled in a critical section, and not the actual writing of the log. Allocating these buffers from the stable memory results in another advantage. Assuming a sufficient amount of stable memory, it is possible to buffer in memory all log records of a transaction, until it is committed. Consequently, only log records of committed transactions are flushed to the stable log, and therefore, the BDB has only committed values. As was already mentioned, having in the BDB only committed values simplifies recovery significantly. Care must be taken, so that sequences of committed log records are flushed to the disk in commit order.

Another possible use of stable memory in our scheme is for storing the recovery record, instead of writing it to disk.

10 Conclusions and Further Research

This paper lays the foundation to an innovative, yet general approach to recovery in MMDBS. Theoretically, the presented scheme can yield almost idealistic performance of transaction processing. However, there are several questions that need further consideration.

As was mentioned in Section 3, the architecture of the log processor and the BDB disks deserves some careful and thorough analysis. The problem of the granularity of updates to the BDB (i.e., individual record updates instead of whole pages) might call for a better solution than the one we outlined. Buffering log records on a page basis similarly to what is done in the page-based algorithm [LC], might be the right direction. Another question to be asked is, given the tremendous I/O bandwidth of schemes like RAID, what is the best way to utilize such power? We are confident that our scheme takes good advantage of this power in the context of transaction processing, but are there better ways?

The discussions in Section 9 indicated that for the purposes of nearly-resident MMDBS, fuzzy and log-driven checkpoints are the appropriate approaches. Further investigation is necessary to determine the trade-off among these two.

When considering the use of stable memory, the page-based algorithm is quite attractive. Again, further research is needed to compare the version of log-driven backups with stable memory, and the page-based algorithm.

The correctness of our scheme does not depend on the memory size ratio between the PDB and the BDB. However, an important question to be asked is under what such ratios our scheme performs best. It is clear that when the PDB is large enough to hold the frequently accessed database pages our scheme performs well. Simulation and experimentation are essential in order to obtain more quantitative statements.

Lastly, we have not discussed the details of loading the PDB into memory after a crash. This has significant impact on recovery time, which is a crucial performance metric.

References

[ANO] Anonymous et al., *A Measure of Transaction Processing Power*, Datamation, Vol. 31, No. 7, April 1985. Also available as Technical Report 85.2, Tandem Computers, Cuperinto, California, 1985.

[BI] *Panel: The Effect of Large Main Memoriy on Database Systems*, Chairperson:, Bitton, D., Panelists: Garcia-Molina, H., Gawlick, D., Lomet, D., Proceedings of the ACM-SIGMOD International Conference on Management of Data, Washington, D.C., May 1986, pp. 337 –339.

[BHG] Bernestein, P. A., Hadzilacos, V., and Goodman, N., *Conçurrency Control and Recovery in Database Systems*, Addison Wesley, 1987.

[DEW] DeWitt, D. J., Katz, R. H., Olken, F., Shapiro, L. D., and Stonebraker, M. R., *Implementation Techniques for Main Memory Database Systems*, Proceedings of the ACM-SIGMOD International Conference on Management of Data, May 1984, pp. 1–8.

[DO] Douglis, F., and Ousterhout, J. K. *Beating the I/O Bottleneck*, ACM SIGOPS Operating Systems Review, Vol. 14, No. 4, October 1988, pp. 26–35.

[EIC1] Eich, M., *Main Memory database Recovery*, ACM-IEEE Fall Joint Computer Conference, 1986, pp. 1226 – 1232.

[EIC2] Eich, M., *A Classification and Comparison of Main Memory Database Recovery Techniques*, Proceedings of the 3rd International Conference on Data Engineering, Los Angeles, California, February 1987, pp. 332–339.

[GMLV] Garcia-Molina, H., Lipton, R. J., and Valdes, J. *A Massive memory Machine*, IEEE Transactions on Computers, Vol. C-33, No. 5, may 1984, pp. 391–399.

[GMS] Garcia-Molina, H., and Salem, K., *System M: A Transaction Processing System for Memory Resident Data*, Princeton University, Computer Science Department, CS-TR-195-88.

[GRY] Gray, J., *Notes on Database Operating Systems*, Operating Systems: An Advanced Course, Lecture Notes in Computer Science 60, Springer-Verlag, Berlin, 1978, pp. 393–481.

[HAG] Hagmann, R. B., *A Crash Recovery Scheme for Memory-Resident Database system*, IEEE Transactions on Computers, Vol. C-35, No. 9, September, 1986, pp. 839–843.

[KIM] Kim, M. Y., *Synchronized Disk Interleaving*, IEEE Transactions on Computers, Vol. C-35, No. 11, November 1986, pp. 978–988.

[LC] Lehman, T. J., and Carey, M. J., *A Recovery Algorithm for a High-performance Memory-Resident database System*, Proceedings of the ACM-SIGMOD International Conference on Management of Data, San Francisco, California, May 1987, pp. 104–117.

[LMP] Lampson, B. W., *Atomic Transactions.* In Distributed Systems — Architecture and Implementation: An Advanced Course, Springer Verlag, Berlin, 1981, Chapter 11, G. Goos and J. Hartmanis (eds.), pp. 246–265.

[LN] Li, K., and Naughton, J. F., *Multiprocessor Main Memory Transaction Processing*, Proceedings of the International Symposium on Databases in Parallel and distributed Systems, Austin, Texas, December 1988, pp. 177–187.

[PB] Park, A., and Balasubramanian, K.,*Providing Fault Tolerance in Parallel Secondary Storage Systems*, Princeton University, Computer Science Department, CS-TR-057-86.

[PGK] Patterson, D. A., Gibson, G., and Katz, R. H., *A Case for Redundant Arrays of Inexpensive Disks (RAID)*, Proceedings of the ACM-SIGMOD International Conference on Management of Data, Chicago, Illinois, May 1988, pp. 109–116.

[PU] Pu, C., *On-the-fly, Incremental, Consistent Reading of Entire Databases*, Algorithmica, No. 1, Springer-Verlag, new York, 1986, pp. 271–287.

[SGM1] Salem, K., and Garcia-Molina, H., *Checkpointing Memory-Resident databases*, Princeton University, Computer Science Department, CS-TR-126-87.

[SGM2] Salem, K., and Garcia-Molina, H., *Crash Recovery for Memory-Resident databases*, Princeton University, Computer Science Department, CS-TR-119-87.

[SGM3] Salem, K., and Garcia-Molina, H., *Crash Recovery Mechanisms For Main Storage Database Systems*, Princeton University, Computer Science Department, CS-TR-034-86.

[SGM4] Salem, K., and Garcia-Molina, H., *Disk Striping*, Proceedings of the 2nd International Conference on Data Engineering, Los Angeles, California, February, 1986.

[SKPO] Stonebraker, M. R., Katz, R. H., Patterson, D. A., Ousterhout, J. K., *The design of XPRS*, Proceedings of the 4th International Conference on Very Large Data Bases, Los Angeles, California, August 1988, pp. 318–330.

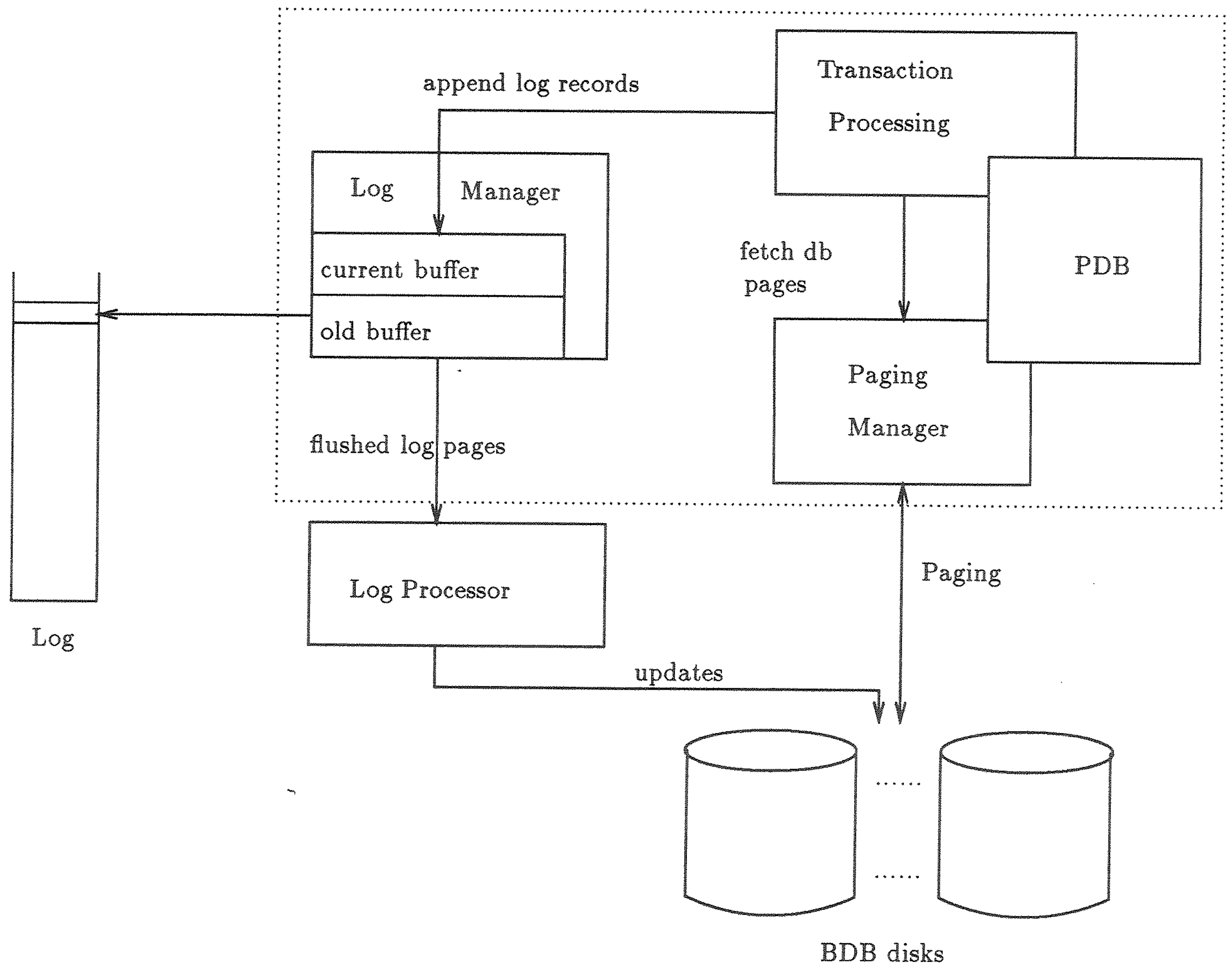
append log records
Transaction Processing
Log Manager
current buffer
old buffer
fetch db pages
PDB
Paging Manager
flushed log pages
Log Processor
Paging
Log
updates
......
......
BDB disks

Figure 1

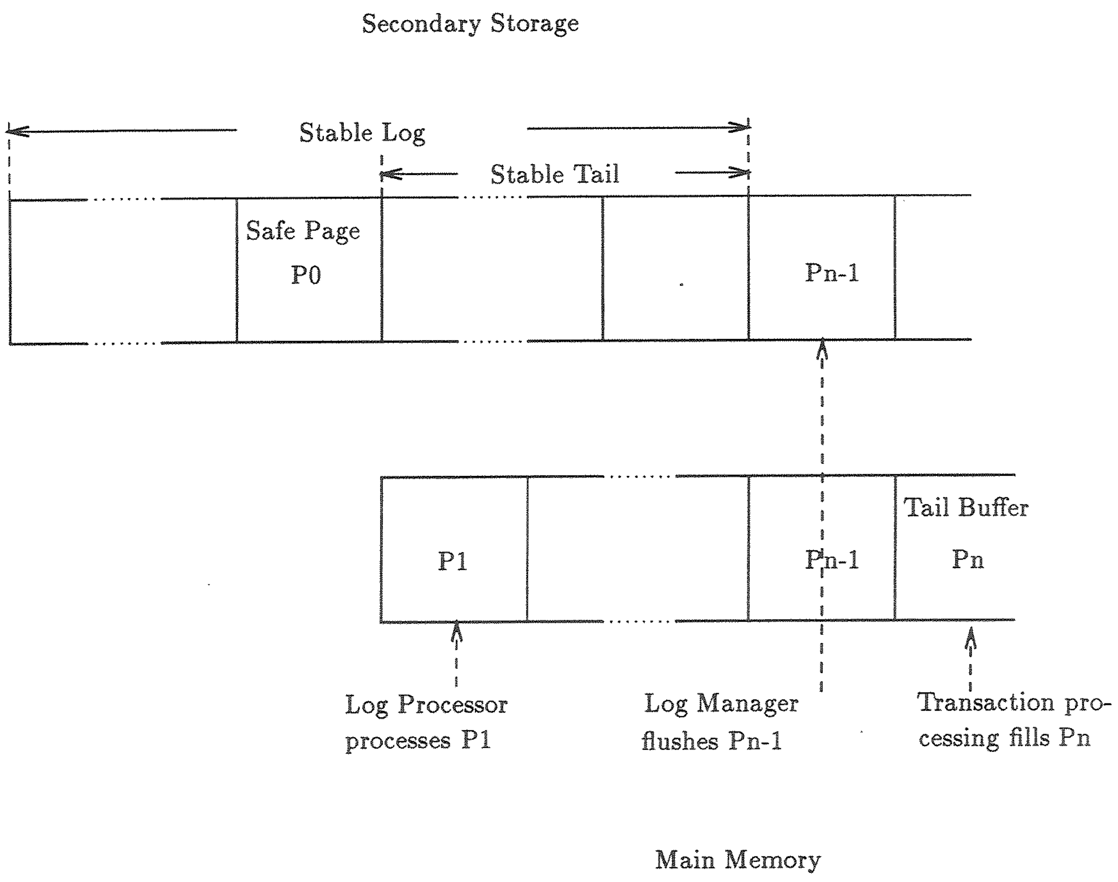
Secondary Storage
Stable Log
Stable Tail
Safe Page
P0
Pn-1
P1
Pn-1
Tail Buffer
Pn
Log Processor processes P1
Log Manager flushes Pn-1
Transaction processing fills Pn
Main Memory

Figure 2