Recovering from Main-Memory Lapses

H.V. Jagadish Avi Silberschatz S. Sudarshan

AT&T Bell Labs.
600 Mountain Ave., Murray Hill, NJ 07974
{jag,silber,sudarsha}@allegra.att.com

Abstract

Recovery activities, like logging, checkpointing and restart, are used to restore a database to a consistent state after a system crash has occurred. Recovery related overhead is particularly troublesome in a main-memory database where I/O activities are performed for the sole purpose of ensuring data durability. In this paper we present a recovery technique for main-memory databases, whose benefits are as follows. First, disk I/O is reduced by logging to disk only redo records during normal execution. The undo log is normally resident only in main memory, and is garbage collected after transaction commit. Second, our technique reduces lock contention on account of the checkpointer by allowing action consistent checkpointing — to do so, the checkpointer writes to disk relevant parts of the undo log. Third, the recovery algorithm makes only a single pass over the log. Fourth, our technique does not require the availability of any special hardware such as non-volatile RAM. Thus our recovery technique combines the benefits of several techniques proposed in the past. The ideas behind our technique can be used to advantage in disk-resident databases as well.

1 Introduction

Current computer systems are able to accommodate a very large physical main memory. In such an environment, it is possible, for certain type of applications, to keep the entire database in main memory rather than on secondary storage. Such a database system is referred to as a main-memory database (MMDB). The potential for substantial performance improvement in an MMDB environment is promising, since I/O activity is kept at minimum. Because of the volatility of main memory, updates must be noted in stable storage on disk in order to survive system failure. Recovery related processing is the only component in a MMDB that must deal with I/O, and hence it must be designed with care so that it does not impede the overall performance.

The task of a *recovery manager* in a transaction processing system is to ensure that, despite system and transaction failures, the consistency of the data is maintained. To perform this task, book-keeping activities (e.g., checkpointing and logging) are performed during the normal operation of the system and restoration activities take place following a failure. Logging notes on stable storage all updates done to the database, and checkpointing periodically creates a consistent snapshot of the database on disk. When a system is restarted after a system crash, recovery activities have to be performed first, and transaction processing can proceed only after necessary recovery activities are performed. To minimize the interference to transaction processing caused by recovery related activities, it is essential to derive schemes where the length of time it takes to do a checkpoint, as well as the time to recover from system failure are very short. It is the aim of this paper to present one such scheme, tailored to main-memory databases.

For simplicity we assume that the entire database is kept in main memory, while a backup copy is kept on disk and is only modified when a checkpoint takes place. However, the ideas behind our technique can be used profitably in disk resident databases as well, where parts of the database may need to be flushed to disk more often in order to make space for other data. A checkpoint dumps some fraction of the database residing in main memory onto the disk. A write-ahead log is also maintained to restore the database to a con-

sistent state after a system crash. The key features of our scheme are as follows:

- The write-ahead log on disk contains only the redo records of committed transactions; this minimizes recovery I/O. We maintain in main memory the redo and undo records of active transactions (i.e., transactions that have neither committed nor aborted). Undo records of a transaction are discarded once the transaction has committed. Undo as well as redo records of a transaction are discarded once it has has aborted. The undo records of a transaction are written to disk only when a checkpoint takes place while the transaction is active. By writing out undo records thus, we are able to perform checkpointing in a state that is action consistent but not transaction consistent.[1]

- The recovery actions after a system crash make only a single pass over the log. The usual backwards pass on the log to find 'winners' and 'losers' and undo the actions of losers is avoided by keeping the undo log separate from the redo log. Recovery is speeded up significantly by reducing I/O in case the redo log does not fit in main-memory. (Although the redo and undo records of transactions that are active at any given point of time can be expected to fit in memory, we do not assume that the *entire* redo log fits in memory.)

- Our technique can be used with physical as well as logical logging.

- A checkpoint can take place at almost any point (namely, in any action consistent state) and the database can be partitioned into small segments that can be checkpointed separately. Interference with normal transaction processing is thereby kept very small.

- No assumptions are made regarding the availability of special hardware such as non-volatile RAM or an adjunct processor for checkpointing. Consequently, the scheme proposed here can be used with any standard machine configuration.

The area of recovery for main-memory databases has received much attention in the past. We present the connections of the present work to earlier work in the area, in Section 8.

The remainder of this paper is organized as follows. In Section 2 we present our system model. In Section 3 the basic checkpoint and recovery scheme is presented. The correctness of this scheme is established in Section 4. Various extensions to the basic scheme, including the segmentation of the database and logical logging, are presented in Sections 5, 6 and 7. Related work is described in Section 8, and in Section 9 we discuss miscellaneous aspects of our technique. Concluding remarks are offered in Section 10.

2 System Structure

In this section we present the system model used in this paper, and describe how transaction processing is handled.

2.1 System Model

The entire database is kept in main memory, while a backup copy, possibly out of date and not transaction consistent, is kept on disk. We assume that disk storage is stable and will never lose its content. For instance, disk mirroring or RAID architectures may be used to ensure this, but the specific disk replication strategy is orthogonal to our discussion here. The system maintains a redo log on the disk, with the tail of the log in main memory. Information about actions that update the database, such as writes, is written to the redo log, so that the actions can be redone if required when restarting the system after a crash. At various points in time the tail is appended to the log on the disk. We refer to the portion of the redo log on the disk as the *persistent redo log* (or as the *persistent log*) and the portion of the redo log in main memory as the *volatile redo log*. The entire redo log is referred to as the *global redo log* (or as the *global log*).

The global log consists of all the redo records of the committed transactions, and the redo records of a committed transaction appear consecutively in the global log. This is in contrast to traditional logs where the log records of different transactions are intermingled. To achieve this, the redo records of an active transaction are kept initially in a private redo log in main-memory, and these redo records are appended to the global log only when the transaction begins its commit processing. (This aspect of the model is not central to our algorithms, and later we discuss extensions that allow redo

[1] The issue of action consistency is important if logical operation logging is used.

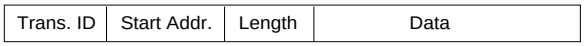

Figure 1: Structure of a Physical Log Record

records to be written directly to the global log tail.) We say that a transaction *commits* when its commit record hits the persistent log. When this occurs, the system can notify the user who initiated the transaction that the transaction has committed.

Initially, we assume that the only actions that modify the database are writes to the database, and writes are logged to the redo log. The structure of a typical physical log record is shown in Figure 1. The transaction id field identifies the transaction, the start address and length specify the start and length of a range of bytes that have been modified, and the value field stores the new byte values of the range of bytes. Later, we consider actions that are encapsulated and treated as a unit for the purpose of logging.

For ease of exposition, we *initially* require that the following condition hold:

Condition LA1: Actions logged are idempotent and are atomic; that is, repetition of the actions in a state where the effect of the actions is already reflected is harmless, and any stable image of the database is in a state after an action finished execution or in a state before the action started execution.

In Section 6 we shall relax this restriction.

Some recovery techniques proposed in the past to do away with undo logging assume deferred updates [4]. Deferred updates require a mechanism to note updates done on an object by an uncommitted transaction without executing them, and redirecting further accesses on the object to the updated copy instead of the original. A mechanism to install the deferred updates after commit is also required. In an object-oriented database, the redirecting of accesses may be particularly troublesome.[2] Our recovery technique does not assume the use of deferred updates (i.e., allows in-place updates), and is thus more general.

The backup copy of the database on disk is updated only when a checkpoint is taken. We allow a checkpoint to be taken at any time, which implies that the backup copy may contain dirty pages — pages that

[2]Shadow paging can remove the look-up overhead by making use of virtual memory address mapping, but carries with it a space cost.

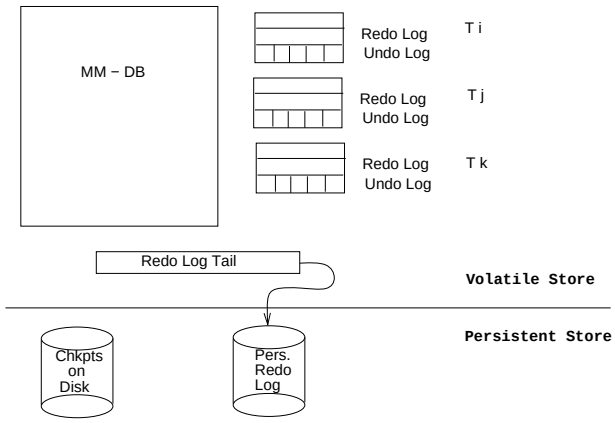

Figure 2: System Model

contain data produced by transactions that have not committed yet. The possibility of having dirty pages on the backup copy implies that that we need to be able to undo the effect of those transactions that were active when the checkpoint took place, and that have since aborted. We do so by keeping in memory, for each active transaction, a private log consisting of all the undo records of that transaction. The private undo log of a transaction is discarded after the transaction either commits or aborts. The undo logs of all the active transactions are flushed to disk when a checkpoint takes place (see Section 3.1). An overview of our system model is presented in Figure 2.

Access to the MMDB is via transactions. Each transaction is atomic and the concurrent execution of these transactions must be serializable. In this paper we assume that serializability is achieved through the use of a rigorous two phase locking (R2PL) protocol, where all locks are released only after a transaction either commits or aborts. The use of the R2PL protocol also ensures that the commit order of transactions is consistent with their serialization order. The granularity of locking is irrelevant to our algorithm; it can be at the level of objects, pages, extents or even the entire database (e.g., if transactions are run serially). Further, our recovery technique permits extended locking and logging modes (such as increment/decrement locks, with corresponding redo/undo log records), provided the schedules satisfy some simple recoverability conditions (which we describe later).

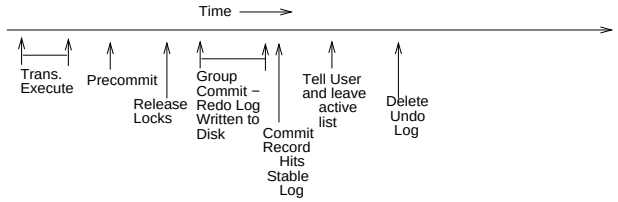

Figure 3: Steps in Transaction Processing

2.2 Commit Processing

When a transaction T_i starts its execution, it is added to the list of active transactions, and the record $\langle start\ T_i \rangle$ is added to the private redo log of T_i. While the transaction is executing, its redo and undo records are maintained in the private logs. When T_i finishes executing, it *pre-commits*, which involves the following steps:

Pre-commit Processing:

1. Transaction T_i is assigned a commit sequence number, denoted by $csn(T_i)$, which is a unique spot in the commit order.

2. Transaction T_i releases all the locks that it holds.

3. Transaction T_i is marked as 'committing' and its commit sequence number is noted in the active transaction list.

4. The record $\langle commit\ T_i, csn(T_i) \rangle$ is added to the private redo log, and the private redo log is appended to the global log. (The commit record is the last log record of a transaction to be appended to the global log.)

Transaction T_i actually commits when its commit record hits the disk. After this has occurred, the system executes the following post-commit processing steps.

Post-commit Processing:

1. Notify the user that transaction T_i committed (and pass back any return values).

2. Remove T_i from the list of active transactions.

3. Delete the volatile undo log created for T_i.

As with other log-based recovery schemes, the persistent log can be considered the *master database*.

Figure 3 outlines the main steps in redo logging and commit processing.

We are in a position to state several key properties of our scheme. Before doing so, however, we need to define the following.

Definition 1: We say that two database states are *equivalent* if they cannot be distinguished by any transactions. The definition accounts for abstract data types that may have different internal structures but that cannot be distinguished by any operations on the abstract data types. □

The following condition ensures that redo logging is done correctly for each transaction:

Condition RP1: Consider the set of objects accessed by a transaction T_i that is executing alone in the system. Suppose that transaction T_i finds this set of objects in state s when first accessed, and its execution takes the set to state s'. Then replaying the redo log of transaction T_i starting from state s takes the set of objects to a state equivalent to s'.

Since the R2PL protocol ensures that the commit order is the same as the serialization order, and since we write out redo records in commit order, the following two key properties hold:

Property CO1: The order of transactions in the persistent log is the same as their serialization order.

Property CO2: The commit order of transactions is the same as the precommit order of transactions. Further, a transaction commits only if all transactions that precede it in the precommit order also commit.

Condition RP1 and Properties CO1 and CO2 ensure that replaying the redo log starting from the empty database (and executing only redo actions of committed transactions) is equivalent to a serial execution of the committed transactions in an order consistent with their serialization order (i.e., the two bring the database to equivalent states). After presenting the checkpointing algorithm, we will discuss ways to recover from a system crash without replaying the entire log.

2.2.1 Discussion

The use of private redo logs reduces contention on the global log tail, as noted in [11]. The log tail is accessed only when a transaction has pre-committed, and repeated acquisitions of short-term locks on the log tail is eliminated. Although writing out private redo records at the end of the transaction can slow down the commit process for transactions that write many log records, it speeds up processing of transactions that write only

a few (small) log records. It is not hard to extend the technique to allow redo records (but not commit records) to be written ahead for large transactions (e.g., whenever there is a pageful of redo records), and ignored on restart if the transaction does not commit.

The release of locks on pre-commit allows a transaction T_i to access data written by an earlier transaction T_j that has pre-committed but not committed. However, T_i has to wait for T_j to commit before it can commit. This is not a problem for updaters (since they have to wait to write out a commit record in any case). However, for read-only transactions that have read only committed data, such a wait is unnecessary. Read-only transactions may fare better under an alternative scheme that holds all locks until commit, since read-only transactions as above can commit without being assigned a spot in the commit sequence order.

The benefits of the two schemes can be combined by marking data as uncommitted when a pre-commit releases a lock, and removing the mark when the data has been committed. Then, read-only transactions that do not read uncommitted data do not have to wait for earlier updaters to commit. Refining the scheme further, uncommitted data can be marked with the commit sequence number of the transaction that last updated the data. A read-only transaction can commit after the commit of the transaction whose csn is the highest csn of uncommitted data read by the read-only transaction.

2.3 Abort Processing

An undo log record (see, e.g. [9]) contains information that can be used to undo the effects of an action. For example, a physical undo log record stores the old value of updated data.

Undo logging is implemented as follows. The undo records are written to the volatile undo log ahead of any modification to memory. The undo log records are not written to disk except when a checkpoint is taken. The undo log records of each transaction are chained so that they can be read backwards. After a transaction commits, the volatile undo log of the transaction may be deleted. (Similarly, the undo log may be deleted after a transaction aborts — see the description of abort processing below.)

We require the following condition on undo logs:

Condition UL1: The effect of a transaction that has not pre-committed can be undone by executing (in reverse order) its undo log records.

Abort processing is done as follows.

Abort Processing: When a transaction T_i aborts, its undo log is traversed backwards, performing all its undo operations. Each undo action is performed and its undo record is removed from the undo log in a single atomic action. After all the undo operations have been completed, the record $\langle abort\ T_i \rangle$ is added to the global log. The transaction is said to have *aborted* at this point. After a transaction has aborted, it releases all the locks that it held.

We do not require the global log to be flushed to disk before declaring the transaction aborted. Also, since we assumed R2PL, there is no need to reacquire any locks during abort processing. The use of the abort record in the persistent log will be made clear once we discuss the checkpoint scheme.

3 Checkpointing and Recovery

In this section we describe the main details of our checkpointing and recovery scheme. For ease of exposition, we describe first an algorithm for an unsegmented database. This algorithm, however, could cause transactions to wait for an inordinately long time. In Section 5, we address the problem by extending this basic scheme to a segmented database where each segment is checkpointed separately. In such an environment, the length of time for which transactions are delayed is reduced correspondingly.

3.1 Checkpointing

Checkpointing is done in an *action consistent* manner (i.e., no update actions are in progress at the time of the checkpointing). Action consistency implies that the database and the undo log are frozen in an action consistent state during the course of the checkpoint. We discuss alternative ways of implementing freezing after presenting the basic algorithm.

Checkpoint Processing:

1. Freeze all accesses to the database and to the undo log in an action consistent state.

2. Write the following out to a new checkpoint image

   (a) A pointer to the end of the persistent log.

   (b) The undo logs of all active transactions,

   (c) The main-memory database.

(d) The transaction IDs and status information of all transactions that are active at the end of the checkpoint.[3]

(e) The last assigned commit sequence number.

3. Write out the location of the new checkpoint to the checkpoint location pointer on stable store. After this, the old checkpoint may be deleted.

We assume that there is a pointer in stable store to the latest checkpoint. The last action performed during a checkpoint is the update of this pointer. Thus, we follow a ping-pong scheme (see [17]), keeping up to two copies of the database. Partially written checkpoints are ignored in the event of a crash, and the previous (complete) checkpoint is used instead, so the writing of the checkpoint is atomic (i.e., it either happens completely or appears to have not happened at all). The set of active transactions and their status is not changed during the checkpoint.

It is not hard to see that our protocol ensures the following two conditions:

1. The *undo log record* is on stable store before the corresponding update is propagated to the database copy on disk, so that the update can be undone if necessary.

2. Every *redo log record* associated with a transaction is on stable store before a transaction is allowed to commit, so that its updates can be redone if necessary.

3.1.1 Discussion

Although in the above description the main-memory database is written out to disk, it is simple enough to apply standard optimizations such as spooling out a copy to another region of main memory, and writing the copy to disk later, and further optimizing the scheme by spooling using copy on write [16]. These techniques together with the segmented database scheme described later, reduce the time for which the database activities (or accesses to parts of the database, in case segmenting is used) is frozen.

In contrast to most other checkpoint schemes, we do not require the redo log to be flushed at checkpoint time (although we do require the undo log to be checkpointed). As a result the (backup) database on disk may be updated before the redo log records for the corresponding updates are written out.

However, the checkpoint processing algorithm makes the following guarantee: any redo records that occur in the persistent log before the pointer obtained above have their effects already reflected in the database.[4] Thus, they need not be replayed (and are not replayed). But commit records do not have this consistency guarantee, since the status of active transactions may still need to be changed. There may be redo operations reflected in the database, but that appear after the persistent log pointer in the checkpoint. We describe later how to handle both cases in the recovery algorithm.

Some checkpointing schemes such as that of Lehman and Carey [11] require checkpoints to be taken in a transaction consistent state, and the redo log to be flushed to disk at checkpoint time. However, transaction consistent checkpoints can lead to lower concurrency and a longer checkpoint interval, especially if long transactions are executed.

To implement freezing of access in an action consistent manner, we can use a latch that covers the database and the undo log. Any action has to acquire the latch in shared mode at the start of the action, and release it after the action is complete. The checkpointer has to acquire the latch in exclusive mode before starting the checkpoint, and can release it after checkpointing is complete.

Action consistency is not really required in the case of physical logging, since the undo/redo action can be performed even if a checkpoint was made at a stage when the action was not complete. However, we require the following:

Condition UL2: Insertion of records into the undo log does not occur during checkpointing.

The above condition ensures that the undo log written at checkpoint time is in a consistent state.

3.2 Recovery

Unlike most other recovery algorithms, our recovery algorithm is essentially one pass, going forward in the persistent log. The recovery scheme is executed on restart

[3] Although we assume here that access to the database is frozen, we relax the assumption later.

[4] In case the algorithm is modified to write out redo records to the global log before pre-commit, care must be taken to ensure this condition.

after a system crash, before the start of transaction processing, and it consists of the following:

Recovery Processing:

1. Find the last checkpoint.
2. From the checkpoint read into main memory:
   (a) The entire database.
   (b) The pointer to the end of the persistent log at checkpoint time.
   (c) The transaction IDs and status information of all transactions that were active at checkpoint time.
   (d) The undo logs of all transactions active at checkpoint time.
   (e) The last assigned commit sequence sequence number at checkpoint time.
3. Go backward in the persistent log from the end until the first commit/abort record is found. Mark the spot as the end of the persistent log.
4. Starting from the persistent log end noted in the checkpoint, go forward in the log, doing the following:

   A.If a redo operation is encountered, Then
   If the operation is a physical redo operation,
   Then Perform the redo operation
   Else /* Steps to handle logical redo
   operations are discussed later */
   B.If an abort record is encountered, Then
   If the transaction was not active at the time
   of checkpoint
   Then ignore the abort record.
   Else find checkpoint copy of (volatile) undo log
   for the transaction, and
   perform the undo operations as above.
   C.If a commit record is encountered,
   Then read its commit sequence number and
   update the last commit sequence number.

5. Perform undo operations (using the checkpointed undo log) for all those transactions that were active at the time the checkpoint took place, and whose commit records were not found in the redo log, and that are not marked committing. After performing the undo operations for a transaction, add an abort record for the transaction to the global log.
6. Perform undo operations (using the checkpointed undo log), in reverse commit sequence number order, for all transactions that were active at the time of checkpoint such that (i) their commit records were not found in the redo log, and (ii) they are marked committing and their commit sequence number is greater than the commit sequence number of the last commit record in the log. After performing the undo operations for a transaction, add an abort record for the transaction to the global log.

We need to skip any redo records at the end of the persistent log that do not have a corresponding commit record. In our implementation, instead of traversing the redo log backwards to skip them, we only go forward in the log, but we read all the records for a transaction (these are consecutive in the log) before performing any action. If the commit or abort record for the transaction is not found, we ignore the log records of the transaction that were read in earlier. Thereby, we avoid the need for atomic writes of individual log records (i.e., log records can cross page boundaries), and the need to keep back pointers in the log. The checkpointed undo log can be expected to fit in main-memory since it only contains undo information for transactions that are active at the time of checkpoint.

Our scheme executes undo operations of a transaction only if the transaction did not commit. A somewhat simpler alternative is to perform undo actions for all the undo log records in the checkpoint first and then perform the redo actions for the committed transactions in the log. With such a scheme there is no longer the need to log abort records for aborted transactions. However, we suffer the penalty of first undoing and then redoing the actions for most transactions active at the time of the checkpoint (which eventually commit). Since this undo followed by redo may be large and expensive, we have used the slightly more complex scheme above where the undos are performed only when determined to be necessary.

By the use of commit sequence numbering, our recovery algorithm can find the transactions that have committed without looking at the commit records in the persistent log preceding the pointer. Alternative schemes, such as using the address of the record in the persistent log instead of the commit sequence number can also be used to similar effect. There may be redo records after the persistent log pointer stored in the checkpoint, whose effect is already expressed in the checkpointed database. Replaying, on restart, of such log records is not a problem for physical log records due to idempotence. When we discuss logical logging we describe how to avoid replaying logical log records whose effect is already expressed in the checkpoint.

This completes the description of the basic version of our recovery scheme. In following sections we will extend the functionality of the recovery scheme to al-

low segmentation of the database and logical logging. First, however, we establish the correctness of the basic recovery scheme.

4 Correctness

The following theorem is the main result that shows the correctness of our recovery scheme.

Theorem 4.1 *If rigorous two-phase locking is followed, recovery processing brings the database to a state equivalent to that after the serial execution, in the commit order, of all committed transactions.* □

Proof: The redo log notes the points at which transactions committed or aborted. Actual undo operations are stored in the checkpoint image. We first show that undo actions are carried out correctly during recovery. We do this via the following claims: (a) we correctly undo the actions of every transaction that did not commit before system crash, AND (b) we do not undo the actions of any transaction that did commit before system crash.

To show (a), we need to show that we undo the effects of every transaction whose updates are reflected in the checkpoint, and further we perform the undo actions in the correct order. Consider any action that has dirtied the checkpoint and did not commit. There are four cases.

Abort Finished Before Checkpoint: Such transactions may still be present in the active transaction list. However, since the abort finished, the effects of the transaction have been undone, and the undo log of the transaction must be empty. Hence no further undo actions are carried out.

Abort Finished After Checkpoint But Before Crash: If the transaction started after the checkpoint, it could not have affected the checkpoint, and no undo log for it can be present. Otherwise it must figure in the checkpointed active transaction list. We will find its abort record, and undo its effects starting from the checkpoint state. (If the transaction aborted due to an earlier crash, and its effects were undone on an earlier recovery, an abort record would have been introduced into the global log at the time of the earlier recovery. If any transaction committed afterwards, the abort log would also have been flushed to disk, so we will reexecute the abort before reexecuting actions of any transaction that started after the previous restart.) It is safe to perform the undo operations at the point where the abort record is found since the transaction must have held locks up to that point (again, logically a transaction that aborted due to a crash can be viewed as having aborted at recovery time without releasing any locks).

Did Not Precommit: The transaction did not precommit and did not abort. Hence it must have held all locks to crash time. The effects of all such transactions are undone at the end of recovery. But no two of them can conflict since all held locks till the crash. (Recall that we assume rigorous two-phase locking).

Precommitted But Did not Commit: This means that not all redo records were written out, so the transaction must be rolled back at recovery. We detect such transactions, since they are marked 'committing' but have larger sequence numbers than the last committed transaction. These must have been serialized after the last committed/aborted transaction, and we roll these back in the reverse of the commit sequence number order, after those that did not precommit. Hence their effects are undone in the correct order.

This completes the proof of claim (a).

To prove claim (b) we need to show that if a transaction commit record hit stable store before crash, it is not rolled back. There are again several cases:

Commit Happened Before Checkpoint: It may still be the case that the transaction is in the active transaction list. But then it must be marked 'committing', and its commit sequence number is less than or equal to that of the last one that committed. We will then not roll it back.

Commit Happened After Checkpoint: Even if the transaction is in the active transaction list, we find the commit record while processing the redo log, and hence we will not roll back the transaction.

This completes claim (b).

We have shown that all required undo operations are executed and no others. No undo action is executed more than once, since there is no repetition during recovery, and any undo operation carried out earlier as part of abort processing is deleted from the undo log atomically with the undo action. It then follows from UL1 that undo operations are replayed correctly.

Redo records are written out to disk in the commit order, and are replayed in the commit order, which is also the serialization order since we assumed rigorous 2PL. Hence they are replayed in the correct order. Every redo operation that is not reflected in the checkpointed segment is replayed, since the redo log of each transaction is flushed to persistent log after transaction precommit, while we noted the end of the redo log as of the start of checkpointing. (Some operations already reflected in the checkpoint may be redone.) Every redo operation that hit the persistent redo log before the checkpoint is reflected in the checkpoint, since the transaction must have precommitted. Hence the action can be considered to have been replayed already. Thus we have shown that we, in effect, replay all necessary redo actions in the correct order. Since physical actions logged are all idempotent, this guarantees that the desired database state is reached.

This completes the proof. □

5 Segmenting the Database

Until now we had assumed that the entire database is checkpointed at one time, while all transactions are frozen. Clearly, this could cause transactions to wait for an inordinately large amount of time. To avoid such delay, we divide the database into units that we call *segments*, and checkpoint each segment independently in an action consistent state.

A segment can be a page, or a set of pages. With small segments, checkpointing a segment will probably have overhead comparable to page flushing in a disk-resident database. For our scheme to work, we require the following condition to hold:

Condition AS1: Each database action that is logged, as well as the actions to redo or undo it, access data resident in only one segment.

The above condition is required so that different segments may be checkpointed independently. Otherwise, during restart if a single redo or undo action accesses different segments checkpointed separately, it could see an inconsistent database state.

The various logging, checkpointing, and recovery techniques described earlier can be used with the following changes:

- The undo log of each transaction is split into a set of undo logs, one for each segment it accesses. Since each action affects only one segment, it is straightforward to do so. The undo log records of a transaction are chained together as before, allowing them to be scanned backwards. Redo logging is done as before.

- Checkpointing is done one segment at a time. (There is no requirement that segments are checkpointed in any particular order, although some performance benefits of ordering are discussed later.) To checkpoint a segment, all accesses to the segment are frozen in an action consistent state. For all active transactions, the undo logs corresponding to the segment are written out, instead of the entire undo logs. Instead of a pointer to the database checkpoint in stable store, a table of pointers, one per segment is maintained in stable store, and these are updated when the checkpoint of a segment (or of a set of segments) is completed.

- We use a latch that covers the segment and its undo log to implement action consistent checkpointing of a segment. Any action on the segment has to acquire the latch in shared mode at the start of the action, and release it after the action is complete.

As before, we do not need to flush the redo log when checkpointing a segment, although we do checkpoint the undo log. Thereby, we reduce the time for which access to the segment must be frozen.

We note that the list of active transactions that have updated the segment but have not committed must not change during checkpointing. This is ensured since a per-segment per-transaction undo log has to be created before a transaction updates a segment, and has to be deleted before a transaction aborts.

Recovery can be done as before, but for each segment we ignore the persistent log records before the persistent log pointer in its last checkpoint. If we split the redo log per segment, we can recover some segments early and permit new transactions to begin operating on those segments while recovering other segments. The idea is discussed further in Section 7, where we also discuss how to relax further the requirement that the checkpoint be made in an action consistent state.

The smaller the segment, the less the time for which access to the segment will be restricted during checkpointing. But logged operations must be kept small enough, or segment sizes should be made large enough to ensure that Condition AS1 is satisfied. If a segment

is large, we can use techniques like the black/white copy on update scheme of [15] to minimize the time for which the segment is inaccessible for transaction processing. We do not specify the size of segments used for checkpointing, except that the segments must contain an integral number of pages (the unit of I/O to persistent storage). The choice may be made by the database administrator. Also, segments need not be predefined, and could possibly be extended dynamically to ensure Condition AS1.

A benefit of segmenting the database, noted in [11], is that segments containing hot spots (i.e., regions that are accessed frequently) can be checkpointed more often than other segments. Recovery for such segments would be speeded up greatly, since otherwise a large number of redo operations would have to be replayed for the segment.

6 Logical Logging

Logging of higher level 'logical' actions as opposed to lower level or physical actions such as read/write, is important for at least two reasons (see [9]). First, it can significantly reduce the amount of information in the log. For example, an insert operation may change a significant part of the index, but the a logical log record that says 'insert specified object in index' would be quite small. (In some cases, there could be a tradeoff between recomputation at the time of recovery and extra storage for physical log records.) Second, with most extended concurrency control schemes, such as [18, 2], physical undo logging cannot be used to rollback transactions — an object may have been modified by more than one uncommitted transaction, and a compensating logical operation has to be executed to undo the effect of an operation. For instance, such is the case with space allocation tables, which we cannot afford to have locked till end of transaction.

Conceptually, we view a logical operation as an operation on an abstract data-type (ADT). For example, an index, or an allocation table can be considered an ADT, and operations such as "insert a tuple" or "allocate an object" can be considered as logical operations. We make the following assumption:

LO1: Each logical operation affects exactly one data item (although the data item may be large, for example, an index).

Typically, the ADT performs its own concurrency control scheme internally, which may not be R2PL (and may not even be 2PL). Some form of higher level locking is used to ensure serializability of transactions.

On system restart, our recovery algorithm performs redo and undo operations in serialization order, and omits operations that were rolled back before the checkpoint. We discuss later how to 'repeat history' by performing redo and undo operations in the exact order in which they occurred originally, but in this section we assume redo and undo operations are performed in serialization order. The design of an ADT that uses logical logging must ensure that when performing redo and undo operations as above, (i) each redone operation has the same result as when it originally executed, and (ii) the ADT is brought to a 'consistent' state at the end of restart; that is, a state that is equivalent (in an application specific sense) to one where the operations corresponding to committed transactions are executed in serialization order. Also, the ADT must be able to undo any operation until transaction commit.

We formalize these notions in the full version of the paper, but for an intuitive idea of what these requirements signify, consider the case of a space allocator. The redo log should contain not only data stating that an allocate request was made, but should also contain data that says what the location of the allocated space was (the location is the return value of the allocation operation). When performing a redo, the allocator must ensure that the same location is allocated. Further, the space allocator must be able to undo both allocate and deallocate requests. To undo a deallocate request, the deallocated space should be reallocated, and its value restored, which means the space should not be allocated to any other transaction until the transaction that performs the deallocate commits. At the end of recovery, the state of the allocation information should be such that all space that was allocated as of the end of recovery is noted as allocated, and all that was free is noted as free. The state may not be exactly the same as if only actions corresponding to committed transactions were executed since the exact layout of the tables may be different. But any difference in the layout is semantically irrelevant, assuming that an allocation request (not a redo of an allocation request) may return any space that was free.

6.1 Logical Logging and Rollback

A logical operation may take a good deal of time to complete. To accommodate such logical operations, we

relax assumption LA1 further here, by allowing checkpointing in the middle of a logical operation. To understand how this can be done, logical operations are best understood as multi-level nested transactions (e.g. [9] or [3]).

In order to roll back partially completed logical actions, we create undo logs for the nested transaction. We create redo log records for logical actions and hence do not need to create redo log records for the nested transaction.

The undo log for the nested transaction, with an identifier i, is embedded in the undo log of the main transaction as follows:

1. A "⟨ begin operation i ⟩" is written to the undo log.
2. The undo operations of the nested transaction are written to the undo log.
3. An "⟨ end operation i ⟩" record, with any information necessary for logical undo, is written to the undo log. The nested transaction is said to commit as soon as the "⟨end operation i ⟩" record enters the undo log. The insertion of the log record is done in an atomic fashion.

On system restart, logical redo operations should not be executed repeatedly since they may not be idempotent, and the "⟨ end operation i ⟩" records are used to ensure non-repetition, as described later. We require the following properties of the undo log:

Condition NT1: The effects of the nested transaction that has not committed can be undone by executing (in reverse order) the undo log records of the nested transaction.

Condition NT2: At any point after the commit of a nested transaction, but before the commit of the main transaction, the effects of logical operation i can be undone by executing the logical undo operation specified in the "⟨ end operation i ⟩" record.

Redo logging in the case of logical actions is the same as with physical actions. We now present versions of the abort processing and recovery processing algorithms that work correctly even with logical logging.

Abort Processing-2: When a transaction aborts, its undo log is traversed backwards, performing all its undo operations. If an "⟨ end operation i ⟩" record is encountered, the logical undo operation is performed, and undo actions of the corresponding nested transaction are ignored. Otherwise the undo actions of the nested transaction are executed. In any case, an undo action is performed and its undo record is removed from the undo log in a single atomic action.

After all the undo operations have been completed, the transaction logs an *abort record* in the shared (redo) log. The transaction is said to have *aborted* at this point. (Note, in particular, that it is not necessary to wait for the abort record to reach the persistent log). After a transaction has aborted, it can release all its locks.

The requirement that logical undo actions are performed and the undo record removed from the log in one atomic action essentially says that checkpointing should not take place while these actions are in progress.

It is important that the designer of the ADT ensure that logical undo operations will never run into a deadlock when acquiring (lower level) locks that they need. If such a situation were to arise, another abort may be need to break the deadlock, which can lead to a cycle that leaves the system hung for ever.

6.2 Checkpointing and Recovery

We now present a modification to the checkpoint processing and recovery processing technique given in Section 3.

Checkpoint Processing-2: Checkpoint processing is done as before, except that if a logical action is implemented as a nested transaction, with its own undo log, checkpointing can be done in a state that is action consistent with respect to the nested transaction's actions. Thus, checkpointing need not be suspended for the entire duration of the logical action.

Recovery processing with logical logging differs from recovery processing with physical logging only in the way logical log records are handled. We describe below the relevant steps of the recovery processing algorithm.

Recovery Processing-2:

1. Find the last checkpoint. /* As before */
2. ... as before, read in checkpoint data.
3. ... as before, find end of persistent log.
4. Starting from the persistent log pointer noted in the checkpoint, go forward in the log:

   A.If a redo operation (numbered, say, i)
   is encountered, Then
   If the operation is a physical redo operation,
   Then Perform the redo operation
   Else /* it is a logical action */
   If there is an "end operation i" record
   in the checkpointed undo log,
   Then ignore the redo operation.
   /* the effect of the operation has been
   reflected in the checkpointed segment,
   and it should not be reexecuted. */
   Else
   If there are undo log records from a nested
   transaction for the logical action
   Then execute the undo operations.
   Execute the redo operation.
   /* Executing the redo operation creates
   undo log records as described earlier */
   B.... handle abort records as before.
   C.... handle commit records as before.

5. ... perform undo operations, as before.
6. ... perform undo operations, as before.

The correctness arguments of the scheme with logical logging are similar to the correctness arguments for the scheme with physical logging. The primary additional concern is that we have to prove that at recovery time we do not redo any action whose effect is already reflected in the checkpoint, and that is not idempotent.

Either a record "⟨ end operation i ⟩" is present in the checkpointed undo log, or it is not. In the first case, we do not replay the logical operation, and its effect is already reflected in the checkpointed segment. In the second case, one of two things is possible. Either the operation had not finished at the time of the checkpoint, and by condition NT1, it is safe to use the undo log of the nested transaction corresponding to the logical action to undo any partial effects of the transaction. The recovery algorithm does the undo, and at this stage the state is equivalent to the state (in a serial replay) just after when the action was initially performed. The recovery algorithm then replays the redo action. Hence, at this stage, the redo operation has been correctly replayed, and the database state reflects the execution of the action. The other case is that the operation had finished at the time of the checkpoint. But the absence of the "end operation" record then implies that the transaction must have committed or aborted before the checkpoint, and in either case we could not have found a redo operation in the persistent log after the persistent log pointed in the checkpoint. In any case, the return values of the redone operations are exactly the same as that of the original operations, and the ADT is in a consistent state at the end of recovery.

7 Extensions

In this section we consider several extensions of the algorithms described so far.

7.1 Repeating History

Our algorithm collects redo records for a transaction together, and outputs the redo records in serialization order to the global log. This reduces the contention on the persistent log tail. On the other hand, if logical operations are implemented using non-R2PL locking at lower levels, care has to be taken in implementing the logical operations to ensure that they have the same effect when redone as when they were done originally, although they have been reordered. The idea of *repeating history* [14] sidesteps this problem by presenting the ADT, at recovery time, with exactly the same sequence of operations as originally occurred.

Our algorithm can be modified to repeat history by logging redo operations to the global log in the order in which the occur, and logging undo operations to the global log only when a physical undo actually occurs (also in the order in which the operations take place). Further, if the undo operation cannot be deduced from the redo operation and the database state when it is re-executed, the undo operation has to be logged along with the redo operation. Typically undo operations occur only rarely, so undo records will be written out only rarely, unlike with other recovery schemes. The recovery algorithm works with minor modifications, and still makes only one pass on the persistent log.

7.2 Database Bigger Than Memory

We assumed earlier that the database fits into main-memory. We can relax this assumption by using virtual memory. Alternatively, we could use the checkpointer to flush some dirty segments, in order to make space for

other segments. Doing so may be preferable to writing dirty pages to swap space since we get the benefit of checkpointing with roughly the same amount of I/O. In fact, our algorithm can be used for disk resident databases as well, and will be efficient provided most of the data in use at any point of time fits into main memory. The idea of writing undo logs only when flushing segments that are not transaction consistent can be used in disk-resident databases as well, and our basic algorithm can be used with some minor modifications even in cases where data does not fit into main memory.

7.3 Partitioning The Redo Log

We can partition the redo log across segments (assuming that every log operation is local to a segment). Partitioning the redo log permits segments to be recovered independently, transactions can start executing before all segments have been recovered, and segments can be recovered on demand. To commit a transaction, we write a 'prepared to commit' record to each segment redo log, then flush each segment redo log. After all segment redo logs have been flushed, we can write a commit record to a separate global transaction log; the transaction commits when this record hits stable storage. Abort records are written to each segment redo log and to the global transaction log. During recovery the global transaction log is used to find what transactions committed and what transactions did not commit. To recover a segment, we bring the segment into main memory and use recovery processing as before on it but using its local redo log, and doing either redoing or undoing the actions of the transaction at the point where the 'prepared to commit' or abort log record is found, depending on whether the commit record is in the global transaction log or not.

Lehman and Carey [11] present a redo log partitioning technique where the log tail is written unpartitioned into a stable region of main memory, and later a separate processor partitions the log tail. However, the technique appears to depend on the availability of stable main memory for the log tail.

7.4 Miscellaneous

If a physical operation has already generated an undo log, we can allow it to proceed even during checkpointing. The undo log can be used to undo the data from whatever intermediate state it is in when the segment is written out.

If checkpointing is done cyclically on the segments (i.e., in a round-robin fashion), we can use a bubble propagation scheme to keep segment checkpoints (almost) contiguous on disk. The idea is to all have segment checkpoints contiguous, except for a single bubble. The bubble is used to create a new checkpoint image for the segment whose old checkpoint is just after the bubble. Once the checkpoint is complete, the bubble is moved forward, replacing the old checkpoint of the segment. The bubble can be used to checkpoint the next segment. Since the undo log that is written out with each segment is not of a predetermined size, some fixed amount of space can be allocated for the undo log, and if the log is too big, any excess can be written in an overflow area.

8 Related Work

For a detailed description of the issues related to main-memory databases, and how they differ from disk-resident databases, see [8]. In this section we concentrate on issues related to checkpointing and recovery. There has been a considerable amount of work on checkpointing and recovery schemes for main-memory databases. Salem and Garcia-Molina [16] and Eich [7] provide surveys of main-memory recovery techniques.

Main-memory recovery differs from recovery in disk-oriented database systems in several ways. The most important differences that we exploit in the present paper are (a) dirty segments are not flushed to disk as often as dirty pages in a disk based system, and (b) the redo and undo logs of uncommitted transactions can be kept in memory and modified without incurring any disk I/O. As a result of (b) we are able to modify the logs and write out to disk only what is absolutely needed to be written to disk, and thus reduce log I/O and recovery time. The benefit of (a) is that undo logs of most transactions never need be written to disk, if the transaction runs to completion without any of its dirty pages being written out.

Some of the details of our recovery scheme are similar to those of Lehman and Carey [11]. Both schemes propagate only redo information of committed transaction to the stable log, and both schemes keep the redo log records of a transaction consecutive in the log. Lehman and Carey also support segmented databases with independent checkpointing for each segment, and logical logging.

The most important contribution of our technique is

that it permits the use of redo-only logging while permitting action consistent checkpointing. The benefits of redo-only logging are clear — recovery time is speeded up by eliminating an analysis pass on the log, and undo operations do not have to be replayed. Li and Eich [13] present an analysis that underscores the benefits of not having undo logging. However, previous techniques paid a high price for this benefit, since checkpointing had to be transaction consistent if undo logging was not done. For example, in the algorithm of Lehman and Carey [11], in order to checkpoint a segment, the checkpointer has to obtain a read lock on the segment. This can adversely affect performance in the case of database hot spots, since the checkpointer will cause contention with update transactions. Levy and Silberschatz [12] also require transaction consistent checkpointing, as do the redo/no-undo techniques described in [4], and the EOS storage manager [5].

To avoid the transaction consistency assumption for checkpointing, we allow dirty pages to be written out. Our idea of writing out undo records when checkpointing dirty pages is, as far as we know, novel. Our technique is thereby able to avoid full undo logging, and at the same time not require transaction consistent checkpointing, thus getting benefits of both worlds.

An alternative, proposed by Eich [6], is not to checkpoint the primary copy of the database, but instead to replay redo logs of committed transactions continually on a secondary stable copy of the database, and have transactions execute on the primary copy only. This would double the storage and processing requirements. Moreover, replaying could become a bottleneck, since it is in effect replaying the committed actions of the main-memory database on the disk database, in serialization order, and could require a considerable amount of I/O.

The black/white checkpointing technique of Pu [15], also described in [17], allows action consistent checkpointing, but either requires deferred updates (shadow paging) or undo logging on disk. The disadvantages of requiring deferred updates were discussed in Section 1. Hagmann [10] allows fuzzy checkpointing, that is, does not even require action consistency. However, logical operation logging cannot be supported by his technique, and undo logging is required, which can slow recovery down.

If the database does not fit entirely into main memory, our technique can checkpoint a dirty segment and swap it, in contrast to other techniques, such as that of Lehman and Carey, that require transaction consistent checkpoints.

The algorithms of [11] and [6] *require* stable main-memory. Our algorithms are not dependent on the availability of stable main-memory. This will enable our algorithms to be used on standard workstations without hardware modifications, which is very beneficial. However, if stable main-memory is available, we can use it for storing the log tail, and thereby achieve better performance in a manner similar to [11] and [6].

Unlike other algorithms that we are aware of, we do not require the redo log to be flushed on every checkpoint. This reduces the per-checkpoint overhead and the time taken per checkpoint, which is important when segments are checkpointed individually.

Unlike many other recovery techniques that support logical operations (such as Aries [14]) our technique does not use *Log Sequence Numbers* (*LSN*s). Instead, we log $\langle$ *end operation i* $\rangle$ records in the undo log, and look it up to find whether a logical action has been carried out already or not. An undo of a logical operation also removes the 'end operation' (atomically) from the undo log, so repeated undo operations are also avoided. These operations are feasible in our environment (unlike in a disk-resident database) since the undo log will most probably reside in main-memory.

The removal of undo records from the undo log, and the removal of redo records of aborted transactions can be viewed as a form of garbage collection of the log [4]. The garbage collection can theoretically be done in a disk-resident database as well, but is inefficient since it involves disk reads and is typically not used.

9 Discussion

Recent TPC benchmark numbers from Oracle indicate the benefits of not writing undo records to disk [1]. In the Oracle database system, pages are locked into memory and thereby prevented from being flushed, for the entire duration of certain kinds of transactions ('discrete transactions'). This forces a bound on the number of such transactions that can be executed concurrently. We noted earlier that our scheme can be used for disk-resident databases as well. By allowing pages to be flushed when required, our scheme can provide the benefits of reduced undo logging while allowing flushing of pages to disk whenever required. We would not have to bound the number of such transactions executing concurrently. We therefore expect our technique to

provide significant performance benefits.

There are other techniques that can be used to avoid undo logging [4]. However, all such techniques that we are aware of require transaction consistent checkpointing. We believe that our technique will be significantly better than the others in environments where many transactions access some 'hot' pages/segments, acquire write locks on different objects in the page/segment, and hold the locks for some non-trivial amount of time (for instance, waiting for messages from remote sites as in two-phase commit, or waiting for disk reads). In such an environment, transaction consistent checkpointing of the hot pages/segments would interfere greatly with regular processing since transaction consistent checkpointing would have to acquire a read lock on the entire page/segment. In other environments, we believe, our techniques will be at least comparable to other techniques that do not perform undo logging.

We have completed a prototype implementation of the recovery technique, and will present a performance analysis in the full version of the paper. We expect our recovery scheme to form the core of a main-memory database system currently being implemented at AT&T Bell Labs.

10 Conclusion

With the general availability of dozens to hundreds of megabytes of main memory on relatively inexpensive and widely used systems, it is rapidly becoming the case that many useful database applications today fit entirely (or largely) within the available main memory. A major factor in performance, and almost the sole cause of disk I/O, is the recovery sub-system of the database, responsible for maintaining the durability of the transactions. In this paper we have presented a recovery scheme for main-memory databases that places no special demands on the hardware, and imposes little overhead at run-time, yet promises the ability to perform a fast recovery.

Acknowledgements

We would like to thank Mike Franklin for taking time off writing his thesis to give us feedback on the paper, pointing us to related work, and for very useful discussions. We would also like to thank Alex Biliris, Narain Gehani and Dan Lieuwen for their comments on the paper, and Ken Salem for providing us information about System M.

References

[1] J. Anderson. Data management: Benchmarking facts of life and why Oracle now comes up a winner. *Open Systems Today*, Apr. 1993.

[2] B. R. Badrinath and K. Ramamritham. Semantics-based concurrency control: Beyond commutativity. *ACM Transactions on Database Systems*, 17(1):163–199, Mar. 1992.

[3] C. Beeri, H.-J. Schek, and G. Weikum. Multi-level transaction management, theoretical art or practical need? In *International Conference on Extending Database Technology, Lecture Notes on Computer Science*, volume 303. Springer Verlag, 1988.

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

[5] A. Biliris and E. Panagos. Eos user's guide, release 2.0.0. Technical report, AT&T Bell Labs, 1993. BL011356-930505-25M.

[6] M. Eich. Main memory database recovery. In *1986 Proceedings ACM-IEEE Fall Joint Computer Conference, Dallas*, pages 1226–1232, 1986.

[7] M. Eich. A classification and comparison of main memory database recovery techniques. In *Proceedings of the Third International Conference on Data Engineering, Los Angeles*, pages 332–339, 1987.

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

[9] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, San Mateo, California, 1993.

[10] R. B. Hagmann. A crash recovery scheme for memory-resident database system. *IEEE Transactions on Computers*, C-35(9):839–843, Sept. 1986.

[11] T. J. Lehman and M. J. Carey. A recovery algorithm for a high-performance memory-resident database system. In *Proceedings of ACM-SIGMOD 1987 International Conference on Management of Data, San Francisco*, pages 104–117, 1987.

[12] E. Levy and A. Silberschatz. Incremental recovery in large-memory database systems. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):529–540, Dec. 1992.

[13] X. Li and M. H. Eich. Post-crash log processing for fuzzy checkpointing main memory databases. In *International Conf. on Data Engineering*, pages 117–124, 1993.

[14] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on Database Systems*, 17(1), Mar. 1992.

[15] C. Pu. On-the-fly, incremental, consistent reading of entire databases. *Algorithmica*, (1):271–287, 1986.

[16] K. Salem and H. Garcia-Molina. Crash recovery for memory-resident databases. Technical Report CS-TR-119-87, Princeton University, Computer Science Department, 1987.

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

[18] W. E. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computers*, C-37(12):1488–1505, Dec. 1988.