RECOVERY IN MAIN MEMORY DATABASES

Le Gruenwald[*]
Jing Huang[*]
Dept of Computer Science
University of Oklahoma
Norman, Oklahoma 73019-0631
{gruenwal,jhuang}@mailhost.ecn.uoknor.edu

Margaret H. Dunham[†]
Jun-Lin Lin[†]
Ashley Chaffin Peltier[†]
Dept of Computer Science and Engineering
Southern Methodist University
Dallas, Texas 75275-0122
{mhd,jun,peltier}@seas.smu.edu

Abstract

With Main Memory DataBases (MMDB) the primary copy of the data resides in volatile main memory. Thus an MMDB is more vulnerable to failures than conventional Disk Resident DataBases (DRDB). This plus the fact that MMDB systems are targeted to high throughput applications, such as phone switching databases, complicates recovery. In this paper we provide an overview of the issues associated with MMDB recovery and briefly examine some of the solutions.

1 Introduction

In a *Main Memory Database (MMDB)* all or a major portion of the database is placed in main memory. What is important is the perception of where data resides. With an MMDB the DBMS software is designed assuming the data is memory resident. With a *Disk Resident Database (DRDB)* the DBMS is designed assuming the data is stored on disk and I/O is required to access the data. MMDB systems are targeted to applications which require high throughput and fast response time. Examples of such applications include ATM bank account information, telephone switching, certain airline reservation applications, and real-time systems.

[*]Supported in part by the National Science Foundation under Grant Number IRI-9201596.
[†]Supported in part by the National Science Foundation under Grant Number IRI-9201643.

Figure 1: MMDB Architecture

Telephone switching applications of today require sophisticated switching hardware and software. Switching of phone calls is directed by the contents of a switching database or routing table. Without the use of MMDB systems, such advanced telephone activities as 800 numbers and call forwarding would not be possible. For example, using Advanced Intelligent Network services, a phone customer can dial a five-digit number to access an 800 number. The mapping between the two numbers must be done within few milliseconds. The memory residency assumption provides not only the required speed, but also the predictability of access time (response time) that is needed.

Mobile computing applications of today center around cellular telephones and applications with limited database processing. We envision mobile computing in the next decade to involve more data intensive applications with the ability of mobile users to request the execution of long lived database transactions. While execution of these transactions will be performed at DBMS systems (either DRDB or MMDB) in the fixed network, the management of these transactions including communicating transaction results to the user at a mobile unit will be performed by a *Base Station*, also called a *Mobile Support Station* [4]. The data needed to manage these mobile transactions will reside in an MMDB at the Base Station to provide fast response. Again, an MMDB provides the speed and predictability needed to access this data.

Figure 1 presents a general MMDB architecture. The MMDB is composed of the *Main Memory (MM)* implemented via a standard RAM memory, and an (optional) nonvolatile memory, *Stable Memory (SM)*. The primary copy of the database resides in MM, and transaction execution is hence performed in MM. The stable (nonvolatile) memory is used to hold log buffers, and avoid I/O actions when transactions are committed. Stable memory is essential to performance in MMDBs [6, 21]. A logger process flushes the logs asynchronously to the log disk. The tail end of the log is in stable memory; enough stable memory to hold at least the log information for active transactions is assumed in this architecture. Archive Memory

(AM) holds a backup of the entire database.

Due to the volatility of RAM memory, recovery of MMDBs has been an area of active research [7, 14, 18]. The objective of this paper is to provide an overview of MMDB recovery. We focus on logging, checkpointing, and reloading. Logging creates a log of update activities to databases. Checkpointing copies the database residing in MM to the AM for backup purposes. Reloading reloads the backup database from AM into MM if a system or media failure occurs that causes the contents of MM to be lost. In the event of system failure a consistent state of the database can be recovered by applying information from the log to an earlier backup (checkpoint) copy of the database. Checkpointing provides a way to limit the work necessary at system restart by reducing the amount of log which must be examined [1, 15]. Hence database logging, checkpointing, and reloading play an important role to ensure that the MMDBs can be recovered from any failure.

The organization of the paper is as follows. The next three sections examine MMDB logging, checkpointing and reload activities. In Section 5 we examine recovery as implemented in several MMDB systems. Finally, conclusion is given in Section 6.

2 MMDB Logging

To keep track of what happens in the database, several types of log records are required in the log (see Table 2). When the MM is modified a record reflecting the prior value of the data, *"BFIM"*, must be written as well as a record reflecting the new value of the data, *"AFIM"*. Transaction status must also be stored in the log. A transaction writes the *"BT"* (*Begin Transaction*) and *"ET"* (*End Transaction*) records to the log to indicate the point when the transaction starts and when it commits. There are two values for "ET" log records: *"commit"* and *"abort"*, representing whether a transaction has been terminated successfully or abnormally. The checkpointer writes *"BC" (Begin Checkpoint)* and *"EC" (End Checkpoint)* records with a special checkpoint identifier *ckptid* to indicate when the checkpoint started and when it ended. A complete checkpoint occurs when a BC record has in the log a paired EC record with the same ckptid.

The information in the log can be collected at different levels of abstraction, mainly physical logging and logical logging. *Physical logging* means the state of the database modified by an operation are logged, and only physical units of storage (Page number, Offset, Length) are known [35]. *Logical logging* contains descriptions of higher level operations and records the state transition of the database [1]. For example, a

Table 1: Log Format and Log Types

Type	**TID**	**LAddr**	**Value**
AFIM	tid	laddr	afim
BFIM	tid	laddr	bfim
BT	tid	$\emptyset$	$\emptyset$
ET	tid	$\emptyset$	commit or abort
BC	ckptid	$\emptyset$	$\emptyset$
EC	ckptid	$\emptyset$	$\emptyset$

log record in logical logging may say "insert record r in relation R and the indices are updated to reflect the modification" [1]. The advantage of logical logging is that it can exploit the semantics of operations and fewer log items are needed to record modifications. However, this advantage is acquired at the expense of added complexity to the database both during normal processing and at system restart. By logical logging, applying the log record "insert record r" twice implies that record r is inserted in the database twice, which produces a different result from applying it once. As the idempotent property does not hold for logical logging, to guarantee the atomicity of transactions the database must make sure that each committed action has been executed exactly once and each uncommitted action has been undone exactly once as well.

With physical logging, the *before images (BFIM)* and the *after images (AFIM)* of updated items are written to the log when the action occurs in the database. As before and after images in the log records reflect the state of the database, applying the BFIMs and AFIMs to the designated parts of the database are idempotent. That is, one can "execute" these "state installation" operations more than once without changing the result. This property can greatly simplify the recovery processing.

The most popular technique for checkpointing MMDBs (fuzzy checkpointing) is best implemented with physical logging. Thus physical logging is recommended for MMDB systems. This is contrary to DRDB where logical logging is usually recommended (because of its reduced space), but physical logging is often used in production systems (due to the ease of implementation).

One of the most accepted types of logging rules is that of *Write Ahead Logging (WAL)* [1, 11, 13, 31], in which log data must be written to a nonvolatile memory prior to the updating in the database. The purpose of the "WAL" protocol is to ensure the effect of uncommitted transactions can be undone [11]. Since only UNDO information of updated items is used to remove the effect of uncommitted transactions, only UNDO information need obey this logging rule [15].

Another important logging rule concerns committed transactions. Once a transaction is committed in a database, the DBMS then must always reflect all of its effect regardless of failure. Thus if a DBMS allows a transaction to commit, the REDO information that was written by this transaction should be ensured in nonvolatile storage. By the commit rule, the recovery procedure is capable of recovering any transactions that completed successfully but whose updates were not physically written to nonvolatile database before the failure occurs in the system.

Both the WAL and Commit rule exist with DRDB. With MMDB systems, a new rule, LAW, is recommended. The idea of this *"LAW" (Logging After Writing)* protocol is that the after image of an updated item should be written to the log after its corresponding update is propagated to the database. Thus if an updating action occurs after the initiation of the last complete checkpoint, the REDO information is guaranteed to follow its begin checkpoint record in the log. Therefore the log portion starting from the begin checkpoint record contains all the log records whose actions may be redone after a crash. This simple idea significantly simplifies the log processing with a fuzzy checkpointing MMDB as demonstrated in Section 3.2.

A nonvolatile log buffer is crucial for high performance MMDBs [6, 21]. This facilitates immediate commit of database transactions without requiring I/O for log records. In addition, the logging rules discussed above can be easily implemented by properly adjusting the order of update and logging.

Briefly speaking, MMDB logging differs from DRDB logging in three ways:

- A nonvolatile log buffer should be used to satisfy WAL without requiring I/O prior to transaction commit.
- Physical logging is recommended as it is easier to use with fuzzy checkpointing.
- To reduce the amount of the log needed to redo transactions after a system failure, the LAW policy should be followed.

3 Checkpointing MMDBs

Checkpointing in MMDBs focuses on low-interference with normal transactions, and supporting efficient recovery. In what follows we review previous research on checkpointing MMDBs. We divide the checkpointing approaches for MMDBs into three groups: fuzzy checkpoint, non-fuzzy checkpoint, and log-driven

checkpoint.

3.1 Fuzzy Checkpointing

Hagmann first suggested using fuzzy checkpointing for MMDBs [16]. The checkpointer does not need to obtain the locks on the data items to be checkpointed. The database is dumped in *sections* (intervals of pages) [16]. The only synchronization between the checkpointer and normal transactions is, after dumping a section, the checkpointer writes a log record to the log. A section must not overwrite its previous image so that if system failure occurs when dumping, its previous image will not be destroyed. This is also called *sliding monoplexed backups* in [32].

Salem and Garcia-Molina compared the fuzzy checkpointing scheme with two non-fuzzy checkpointing schemes, and found that fuzzy checkpointing is the most efficient one [33]. Their fuzzy checkpointing scheme maintains two complete backup databases on disks in addition to the primary database on main memory. The checkpoint process uses a *ping-pong* scheme to flush dirty pages from the primary database to the backup databases. That is, the checkpointer flushes dirty pages to one backup database during the current checkpoint, and flushes dirty pages to the other backup database during the next checkpoint, and so on. Thus, each page requires two dirty bits, and each dirty page is flushed twice, once to each backup database, on two consecutive checkpoints.

Lin and Dunham proposed a variant of fuzzy checkpointing scheme, called *Segmented Fuzzy Checkpointing* [27, 26]. This approach checkpoints one segment at a time in a round-robin fashion and automatically changes the segment boundaries based on the distribution of update operations. As a result, the average recovery time is greatly reduced at the cost of a more sophisticated checkpointing process. The amount of the log needed to read during recovery after a system failure is less with the segmented approach than with the conventional fuzzy one.

Li *et al* proposed a fuzzy checkpointing scheme that uses multiple log disks to reduce recovery time [25]. The database is divided into *partitions*, each of which has its own log disks. The log records generated due to updating the data in a partition, are written to the log disk for that partition. By distributing log records in multiple log disks according their partitions, the time to recover from a system failure is reduced.

3.2 Use of the LAW Protocol with Fuzzy Checkpointing

Figure 2 shows a checkpointing action and two database actions made to the page P_1 of MM. At time t_1 a record in P_1 is deleted, at time t_2 the MMDB flushes P_1 to AM, and at time t_3 a new record is inserted into P_1. A crash occurs before the MMDB gets a chance to copy out P_1 in the most recent checkpoint which is thus an incomplete checkpoint. The delete action has been reflected in AM, but the insert action is not reflected in AM. Therefore, the actions made after the initiation of the most recent complete checkpoint may or may not be reflected in the backup database at the time of failure.

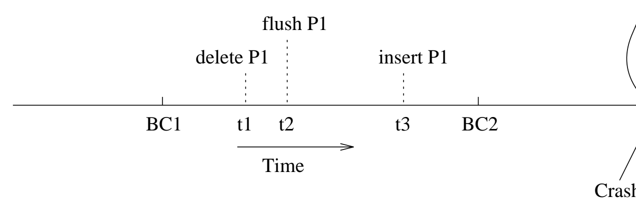

Figure 2: The Database Actions With Fuzzy Checkpoint

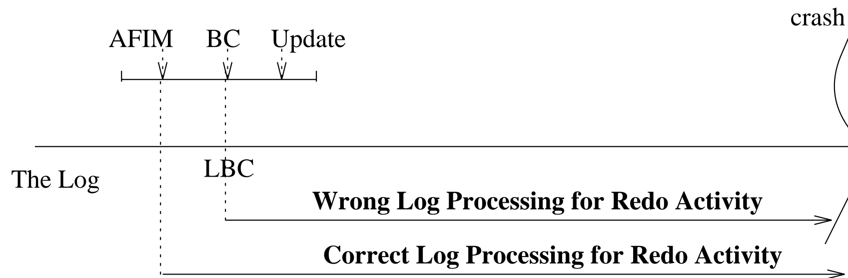

Figure 3: The Post-crash Log Processing Without "LAW"

If the LAW policy is not followed it is possible to have the AFIM record for an update written to the log prior to a BC and the actual update to MM occurring after the point in time when the BC was written. In this case, a proper redo starting point is the earliest beginning of the transactions which are active at the initiation of the last complete checkpoint. ¿From that point, the redo activity is forward processed until the end of the log is reached. Figure 3 demonstrates the correct post-crash log processing with a fuzzy checkpoint MMDB.

The usual approach to identify the redo point is a backward scan of the log from the LBC (Begin Checkpoint record of the last complete checkpoint). This approach certainly reduces the performance of recovery in the MMDB. Furthermore, it becomes worse when there are long-lived transactions in the MMDB. Under the "LAW" protocol, if an action occurs after the LBC, then its after image must follow the

LBC in the log. Figure 4 illustrates this idea. If the log processing for the redo purpose begins immediately from the LBC as shown in Figure 4, it still can redo all actions that should be redone after a crash. With a nonvolatile log buffer, the implementation of the "LAW" protocol is easy. After the MMDB updates the page of MM, it appends the corresponding AFIM record to the log buffer.

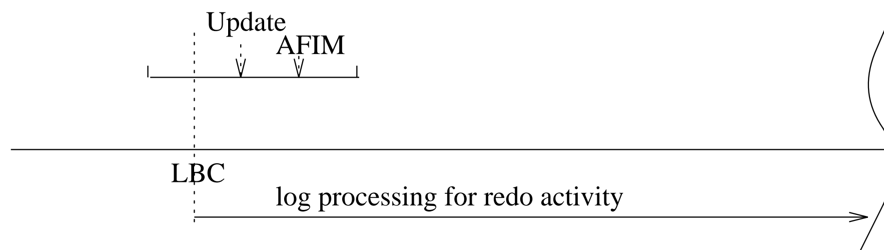

Figure 4: The Post-crash Log Processing Under "LAW"

Thus the use of the LAW protocol reduces the amount of log which must be processed to recover from a system failure.

3.3 Non-Fuzzy Checkpointing

Non-fuzzy checkpointing schemes usually incur much overhead during normal transaction processing. The overhead mainly comes from locking the checkpointed objects to ensure transaction-consistency or action-consistency. Lehman and Carey proposed a transaction-consistent (at *relation* level) scheme, where the main memory is divided into segments, one for each database object (relation, index, or database structure) [21]. The checkpointing transaction locks one relation at a time, checkpoints all related segments of the relation, and then releases the lock. Eventually, every relation in the checkpoint is transaction-consistent; only committed data is checkpointed. Thus, there is no need to maintain undo-log-records in nonvolatile storage. However, the checkpointing activity increases the data contention with normal transactions, and thus greatly affects the performance.

Salem and Garcia-Molina have discussed two non-fuzzy checkpointing approaches in [33]. The first approach aborts some update transactions to ensure the state read by the checkpointing process is consistent. The second approach requires some update transactions storing the original values of data items to be updated for the checkpointing process. Both have severe impact on the system performance.

Jagadish *et al* proposed an action-consistent checkpointing scheme [18]. When checkpointing, the undo-log-records of active transactions are first written to the log, and then dirty pages are flushed to disks to

enforce the WAL protocol. During normal transaction processing, the logger only writes the redo-log-records of the committed transactions to the log. This approach was originally used in Dali, which is discussed in Section 5.1.

3.4 Log-Driven Checkpointing

Instead of dumping from the main memory database, log-driven checkpointing applies the log to a previous dump to generate a new dump. This technique is originally used to generate remote backup of the database [8, 28], and is adopted to MMDBs in [5, 22, 23]. With high transaction processing rate in MMDBs, the size of the log can increase rapidly. As a result, applying the log to the dump could be quite inefficient, compared to flushing the dirty page to the dump directly.

4 MMDB Reloading

In an MMDB system, the primary copy of the database resides permanently in a volatile memory, instead of stable secondary storage as in a DRDB system. When a system or media failure occurs, the database is lost. In order to resume transaction processing quickly without degrading system performance, a reload scheme is needed to bring the database from AM into MM and to construct its most recent consistent state.

There are several issues associated with MMDB reloading. The first issue is *occurrence frequency of the reload process*. System failure typically happens due to power outage, preventative maintenance, DBMS errors, operating system crashes, or hardware downs [19]. Although MM can be made more reliable by techniques such as battery-backed up memory boards, uninterruptible power supplies, and triple modular redundancy, this does not guarantee that the system is failure-free [9]. On average, a system failure occurs once every few weeks [13]. Media failure, although occurring once or twice a year, can have a severe impact on recovery. A memory failure will require either reloading the entire database from AM to MM as in system failure, or reloading a partial database if the specific location of memory failure can be identified. In addition to these failures, if MM is not large enough to hold the entire database, I/Os might also be incurred due to MM page faults. This may happen frequently based upon the percentage of the database in MM. A general reloading scheme should also be capable of handling these page faults.

The second issue is *when the system should resume its execution after a failure*. According to the 80-20

rule, 80% of the accesses go to 20% of the data [?], and thus a transaction can often run with only a small portion of the database present in MM. Therefore, it does not make sense to load the entire AM into MM before bringing the database online. As estimated in [?], 28.43 minutes are needed to recover one gigabyte database. If the system is not available at all during recovery, many transactions will be backlogged. This also translates into many transactions missing their deadlines if the application is deadline-driven.

The third issue is *reload prioritization*. If the system is allowed to be brought up before the entire database is memory-resident, many page faults may be incurred unless data in MM is needed frequently or immediately depending on applications. This means that data should be prioritized for reload purposes. Reload priority can be determined based on access frequency as in [14], transaction deadline or temporal data interval for real-time applications as in [?].

The existing reload schemes can be classified into two categories: *simple reloading* and *concurrent reloading*. In simple reloading, the system can not be brought online until the entire database is memory-resident. This scheme can complete database reloading in the shortest amount of time; however, transaction execution may be blocked for a long time. The techniques proposed in [14, 16] fall under this category. The former resumes transaction processing only after the database has been reloaded completely and log information has been applied to this database to obtain its most consistent state. The latter makes use of the two processors and nonvolatile shadow memory (SM) existing in the MARS system to hasten the recovery process. While one processor reloads the database, the other processor copies AFIMs of committed transactions from the log to SM. The system is brought up as soon as the entire database is memory-resident. When a data item is needed, the system will search for it in both MM and SM. The one found in SM will take precedence over that found in MM. This dual address translation mechanism allows normal transaction processing to be performed even before the consistent state of the database is established.

In concurrent reloading, the system resumes its execution before the entire database is brought into MM. Transaction processing and reload activities can be performed in parallel. In [?, 21, ?] after the system catalogs and their indices are reloaded then regular transaction processing is allowed to resume. When a transaction needs to access relations and indices that are not yet memory-resident, the transaction manager issues recovery transactions to reload them on a per partition basis. Databases entities (tuples or index components) are stored in partitions and do not cross partition boundaries. Since relations and indices are usually of large sizes, transactions must wait for a long time before they can be executed.

Gruenwald has proposed three different concurrent reload algorithms: *Ordered Reload with Prioritization (ORP)*, *Smart Reload (SR)*, and *Frequency Reload (FR)* [14]. The differences between them lie in the structure of AM, utilization of data access frequency, reload prioritization, and reload granularity. A reload of higher priority can preempt that of lower priority. Reload granularity is the smallest unit of data to be reloaded; during the reload of this unit, no preemption can take place. The algorithms distinguish two types of transactions: waiting and executing. Waiting transactions were present in the system but not yet committed at the time of failure. Executing transactions arrive after the system resumes its execution. ORP requires the system to be brought online after some amount of the database is in MM. It gives the highest reload priority to executing transactions, the second highest to waiting transactions, and the lowest to the rest of the database. This algorithm uses cylinder to be its reload granularity; therefore no preemption is allowed to take place until the entire currently reloaded cylinder is brought into MM. SM works similarly to ORP except that whenever it reloads a page, it makes sure that the page with the highest access frequency among all pages on AM is reloaded. Because of this, it chooses block to be its reload granularity. FR is exactly the same as ORP, except that it requires a special AM structure, in which pages are organized on cylinders in decreasing order of access frequency. With this structure, within the reload of one cylinder, more frequently accessed pages are brought into MM before less frequently accessed ones. Simulation results comparing these three algorithms with the simple reload algorithm described above showed that FR yields the best transaction response time and system throughput [14].

Levy and Silberschatz [22] has proposed a concurrent reload scheme that resumes transaction processing immediately after a system failure and recovers pages individually according to the demand of post-crash transactions. It makes use of a Stale/Fresh marking technique. A page is stale if it has been modified by committed transactions but not yet reflected in the disk database, and is fresh otherwise. A transaction that accesses a stale page will trigger the recovery of that page and and will be delayed until the page is reloaded into MM and brought up-to-date. In order to implement a page-based recovery, log records must be grouped together on a page basis during normal operation.

5 Recovery with Existing MMDB Systems

There have been several MMDBs, some of which are commercialized while others are prototyped. This section provides an overview of the recovery techniques implemented in three existing MMDB systems:

Dali from AT&T, Fast Path from IBM, and NEC RTDBMS from NEC America. The first system is a prototype, while the other two are commercial products.

5.1 Dali

Dali is a main memory database server being developed at AT&T Bell Labs [17]. Two Recovery Managers have been implemented for Dali. In the original Recovery Manager, disk I/O was reduced by logging only REDO records during normal execution. Lock contention by the checkpointer was handled by allowing segment-level action-consistent checkpoints and by having the checkpointer write to the disk relevant parts of the UNDO log. Each transaction maintained a private UNDO log for each segment it accessed. Each UNDO log was then checkpointed when its corresponding segment was checkpointed. This allowed the system to quiesce only actions within a segment instead of the actions within the entire database during checkpointing. It also incurred the overhead associated with maintaining several UNDO and REDO logs per transaction, and the global REDO log. The recovery algorithm made only a single pass over the log, and required no special hardware to preserve the data [18].

Tests conducted with the original recovery algorithm used in Dali led to a restructuring of its recovery algorithm [2]. The new features in Dali include multi-level logging, post-commit actions, dirty page detection, and fuzzy checkpoints. Multi-level logging was used to provide higher level of concurrency. Fuzzy checkpoints were implemented due to the poor performance of the original action-consistent ones. Each checkpoint first copies all dirty pages to disk, then takes a snapshot of the active transactions. For every active transaction, all undo logs for that transaction are written out as well. Finally, the global log is flushed and the checkpoint is complete.

5.2 Fast Path

IMS/VS Fast Path supports both memory-resident data and disk-resident data. It performs updates to memory resident data at commit time. As the database is not modified by active transactions, only after images need to be maintained in the log file and no UNDO operations are required when a failure occurs [?, ?]. The idea of group commit is adopted by IMS/VS Fast Path to reduce the amount of time needed to flush log records to log disks during the commit process, which in turn improves system throughput [9].

No writes to the disk database are allowed during normal processing. The transaction-consistent backup copy of the database is refreshed during system shutdown, or some infrequently checkpoints. Two backup

copies of the disk database are maintained. The checkpointed data is written alternately onto two backup copies [?, ?].

At the system restart, the main storage data base (MSDB) is reloaded from one of the backup copies. All the changes made since the last checkpoint will be reflected on the reloaded database by using the log information stored in the log file [?].

5.3 NEC RTDBMS

Main memory database also plays an important role in real-time systems. Two examples are NEC Real-Time Database Management System (NEC RTDBMS) and Stone RTDB [?]. Due to space limitation, in what follows we briefly discuss NEC RTDBMS.

NEC RTDBMS has several important features to ensure high throughput and accurate predictability [?]. First, the primary copy of the database is memory-resident, and thus no page fault occurs during transaction processing. Second, the in-memory log buffer is nonvolatile. This reduces the transaction response time since no log I/O is needed to satisfy the WAL protocol and the Commit Rule. It implements physical logging using deferred update. Third, fuzzy checkpointing is used to minimize the overhead incurred by the checkpointing activity. No real-time characteristics such as transaction deadline and criticalness are utilized in the recovery components.

6 Summary and Conclusion

With increasing capacity and decreasing price of memory, MMDBs become an attractive alternative to DRDBs for applications which require high throughput and fast response time. Since the primary copy of the database in an MMDB system may reside in a volatile memory whose contents are lost upon a system or media failure, recovery is one of the most crucial issues. In this paper, We provided an overview of three recovery components, logging, checkpointing, and reloading, and described how they are implemented in three existing MMDBs: Dali, Fast Path, and NEC.

We discussed three major logging rules: WAL, LAW, and Commit. We also concluded that a nonvolatile log buffer should be used to satisfy WAL without requiring I/O prior to transaction commit and the LAW policy should be followed to reduce the amount of log needed to redo transactions after a system failure.

We described three groups of checkpointing: fuzzy, non-fuzzy, and log-driven, and showed how LAW

can be combined with fuzzy checkpointing to achieve faster recovery. We also identified three major issues associated with reloading in MMDBs: occurrence frequency, when the system should resume its execution after failure, and reload prioritization. An efficient reload scheme should minimize system down time. However, which part of the database must be memory-resident before the system can be brought online must be selected carefully in order to reduce the number of page faults that might be incurred during transaction processing. This means that data should be prioritized for reload purposes.

As many real-time applications need MMDBs to achieve high speed and predictability in response time, future research should investigate how real-time requirements such as transaction deadlines and temporal data intervals can be incorporated into MMDB recovery.

References

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

[2] Philip Bohannon, Rajeev Rastogi, Avi Silberschatz, and S. Sudarshan. Multi-level recovery in the dali storage manager. Technical report, AT&T Bell Labs Internal Report, 1995.

[3] D.L. Burkes and R.K. Treiber. Design approaches for real-time transaction processing remote site recovery. In *35th IEEE Compcon 90*, pages 568–572, 1990.

[4] Margaret H. Dunham and Abdelsalam Helal. Mobile computing and databases: Anything new? *ACM SIGMOD Record*, 24(4):5–9, December 1995.

[5] Margaret H. Eich. Main memory database recovery. In *Proceedings of ACM-IEEE Fall Joint Computer Conference*, pages 1226–1232, November 1986.

[6] Margaret H. Eich. A classification and comparison of main memory database recovery techniques. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 332–339, 1987.

[7] Margaret H. Eich. Main memory databases: Current and future research issues. *IEEE Transactions on Knowledge and Database Engineering*, 4(6):507–508, December 1992.

[8] Hector Garcia-Molina and Christos A. Polyzois. Issues in disaster recovery. In *35th IEEE Compcon 90*, pages 573–577, 1990.

[9] Hector Garcia-Molina and Kenneth Salem. Main memory database systems: An overview. *IEEE Transactions on Knowledge and Database Engineering*, 4(6):509–516, December 1992.

[10] J. N. Gray. Notes on data base operating systems. In *Operating Systems: an Advanced Course*, volume 60, pages 393–481. Springer-Verlag, NewYork, 1978.

[11] Jim Gray. Notes on data base operating systems. In R. Bayer, R.N. Graham, and G. Seegmueller, editors, *Lecture Notes on Computer Science Volume 60*. Springer-Verlag, 1978.

[12] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, 1992.

[13] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, San Mateo, California, 1993.

[14] Le Gruenwald and Margaret H. Eich. Mmdb reload algorithms. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 397–405, 1991.

[15] Theo Haeder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):287–317, December 1983.

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

[17] H. V. Jagadish, D. Lieuwen, R. Rastogi, A. Silberschatz, and S. Sudarshan. Dali: A high performance main memory storage manager. In *Proceedings of the 20th Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA), Santiago, Chile*, August 1994.

[18] H. V. Jagadish, A. Silberschatz, and S. Sudarshan. Recovering from main-memory lapses. In *Proceedings of the 19th Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA), Dublin*, August 1993.

[19] T. J. Lehman and M. J. Carey. A recovery algorithm for a high performance memory-resident database system. In *19 ACM SIGMOD Conf. on the Management of Data*, May 1987.

[20] T. J. Lehman, E. J. Shekita, and L. F. Cabrera. An evaluation of starburst's memory resident storage compoenent. *IEEE Transactions on Knowledge and Database Engineering*, 4(6):555–566, 1992.

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

[22] E. Levy and A. Silberschatz. Incremental recovery in main memory database systems. *IEEE Transactions on Knowledge and Data Engineering*, 4(6), December 1992.

[23] K. Li and J. F. Naughton. Multiprocessor main memory transaction processing. In *Proceedings of International Symposium on Databases in Parallel and Distributed Systems*, December 1988.

[24] X. Li and M. H. Eich. A new logging protocol: LAW. Technical Report CSE9221, Southern Methodist University, Dept. of Computer Science and Engineering, Dallas (TX), October 1992.

[25] X. Li, M.H. Eich, V.J. Joseph, Z. Gulzar, C.H. Corti, and M. Nascimento. Checkpointing and recovery in partitioned main memory databases. In *Proc. IASTED/ISMM International Conference on Intelligent Information Management Systems, Washington, DC.*, June 1995.

[26] J. L. Lin and M. H. Dunham. Dynamic segmented fuzzy checkpointing for main memory databases. *Submitted for publication.* , October 1995.

[27] J.L. Lin and M.H. Eich. Segmented fuzzy checkpointing for main memory databases. Technical Report CSE9442, Southern Methodist University, Dept. of Computer Science and Engineering, Dallas (TX), December 1994.

[28] Jim Lyon. Tandem's remote data facility. In *35th IEEE Compcon 90*, pages 562–567, 1990.

[29] C. Pu. On-the-fly, incremental, consistent reading of entire databases. *Algorithmica, Springer Verlag Inc.*, 1(3):271–287, 1986.

[30] Rajeev Rastogi. personal communication, July 1995.

[31] K. Rothermel and C. Mohan. Aries/nt: A recovery method based on write-ahead logging for nested transactions. In *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, pages 337–346, 1989.

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

[33] K. Salem and H. GarciaMolina. Checkpointing memory-resident databases. In *Proc. IEEE CS Intl. Conf. No. 5 on Data Engineering, Los Angeles*, February 1989.

[34] Kenneth Salem and Hector Garcia-Molina. Checkpointing memory-resident database. In *Proceedings of the Fifth IEEE International Conference on Data Engineering*, pages 452–462, February 1989.

[35] Joost S. M. Verhofstad. Recovery techniques for database systems. *ACM Computing Surveys*, 10(2):167–195, June 1978.