

**RECOVERY SYSTEM FOR
MAIN-MEMORY DATABASE**

CS764 PROJECT REPORT

ANURAG GUPTA, HAN-YIN CHEN

{anurag , han-yin}@cs.wisc.edu

**Computer Sciences Department
University of Wisconsin, Madison
Madison – 53706, WI**

May 14, 1999

Abstract

In a Main Memory Database (MMDB) the primary copy of the database resides in volatile main memory. This makes MMDB systems more vulnerable to failures compared to the traditional Disk Resident Databases (DRDB). A backup copy of the database is maintained in secondary storage for recovery purposes. Recovery activities like logging, checkpointing and reloading are used to restore the database to a consistent state after a system crash has occurred. In this paper, we implement a recovery scheme with action consistent checkpointing for a main-memory database system and measure the performance of the recovery mechanism as a function of the number of tuples and the number of fields in a tuple. We find that the recovery time increases almost linearly with the increase in the size of the database and that the number of tuples in the database affects the recovery time more than the size of an individual tuple.

1. Introduction

In a Main Memory Database (MMDB) system, the primary copy of the database resides in volatile main memory as opposed to Disk Resident Database (DRDB) systems where the database is on disk. The perception of where the data resides is important here. With an MMDB the DBMS software is designed assuming that the data is memory resident. With a DRDB the DMBS software is designed assuming the data is stored on disk and I/O is required to access the data [1]. MMDB systems need not perform I/O operations for database applications. Thus, they are ideal for applications that require high throughput and a fast response time. With the increasing demand for high-performance systems and the steady decrease in memory cost, MMDBs have become an attractive alternative to DRDB's [2]. This is especially the case for telephone switching applications, mobile computing applications and real-time systems, i.e. applications that require high throughput and fast response time [3]. The increasing popularity of MMDBs can be illustrated by the sizable effort of research done in this area [4,5,6,7].

A general MMDB architecture consists of a main memory implemented via standard RAM and an optional non-volatile (*stable*) memory. The main memory holds the primary copy of the database, while the stable memory holds the log and a backup copy of the database. A logger process flushes the logs asynchronously to the disk. Alternatively, an additional nonvolatile memory may be used as a shadow memory. This memory is intended to hold updates performed by active transactions and the tail end of the log that contains information about those updates [2].

While meeting the high-performance demand of many real-time applications, MMDBs are naturally more vulnerable to failures than conventional DRDBs. Thus, the recovery component of an MMDB system must be properly designed, implemented and maintained. Three aspects of the recovery subsystem serve to ensure that the MMDB can recover from any failure: *logging*, *checkpointing* and *reloading*. Logging maintains a log of update activities that occur during normal transaction execution. Checkpointing takes a snapshot of the database periodically and copies it onto stable storage for backup purposes. By reducing the number of log records that must be examined during system restart, checkpointing limits the amount of work necessary to recover from failures. Following a system crash, reloading restores the database to a consistent state. It first reloads into the main memory the backup copy that is recorded during the last checkpoint. A consistent state of the database is then achieved by applying information in the undo and redo logs to the reloaded copy.

This paper is organized as follows. **Section 2** briefly describes the three subsystems of recovery: logging, checkpointing and reloading. It discusses the various flavors of current MMDB recovery algorithms. In **Section 3** and **Section 4** we describe our implementation of the main-memory database and the recovery scheme implemented. We discuss the tests performed to ensure the correctness of the recovery scheme in **Section 5**. In **Section 6**, we present and discuss the performance results. **Section 7** presents some future directions. Finally, we conclude in **Section 8**.

2. Overview of MMDB Recovery techniques

The *recovery manager* in a transaction processing system is responsible for maintaining the consistency of the database system despite system and transaction crashes. To perform this task, book-keeping activities (e.g. logging, checkpointing, etc) are performed during the normal operation of the system. Our discussion of the existing MMDB recovery algorithms is organized around the three pillars of MMDB recovery:

2.1. Logging

In MMDB logging, log information can be recorded at two different levels of abstraction, namely *physical logging* and *logical logging*. Physical logging records, for each update, the state of the modified database and the physical location (*such as page id and offset*) to which the update is applied. Logical logging records the *state transition* of the database and the logical location affected by the update. The advantage of logical logging is that it can exploit the semantics of operation and log fewer items. However in physical logging, the idempotent property of log records is naturally enforced, while in logical logging, measures must be taken to ensure that each committed action is executed exactly once and likewise, each uncommitted action is undone exactly once. Logical logging thus adds extra complexity to the database both during normal processing and during recovery.

The *Write Ahead Logging (WAL)* is the most popular and accepted type of logging rules that is used, in which log data must be written to nonvolatile memory before the update can be made to the database. The WAL protocol ensures that the effect of uncommitted transactions can be undone. For committed transactions, the DBMS must always reflect the updates regardless of failure. Thus, when a transaction commits, the REDO information that was written by this transaction should be forced to nonvolatile storage.

2.2. Checkpointing

Checkpointing reduces the amount of work that must be done during system restart after a DBMS crash. Checkpointing algorithms must be designed such that the interference with normal transaction operations is minimum and that effective recovery can be done after a crash. MMDB checkpointing approaches fall into three categories: *non-fuzzy checkpoint*, *fuzzy checkpoint*, and *log-driven checkpoint*.

In non-fuzzy checkpointing, the checkpointer first obtains locks on the data items to be checkpointed so that consistency of the database is preserved. This consistency, in turn, can be either transaction or action oriented. Action consistency is better and results in less interference than transaction consistency. Clearly a performance price is paid for either choice. In fuzzy checkpointing, the checkpointer does not secure any locks. To synchronize with normal transactions, the checkpointer writes a record to the log after a checkpoint is completed. In addition, the database is generally dumped in segments. In log-driven checkpointing, the checkpointer applies the log to a previous dump to generate a new dump rather than dumping from the main-memory database. This technique is less efficient than flushing the dirty pages directly and is usually used to generate remote backup of the database [3].

2.3. Reloading

Reloading is the last and the one of the most important parts of the recovery process. Reloading must ensure that the database is consistent after a crash. The database is loaded from the last checkpointed image and the undo and redo operations are applied to restore it to the most recent consistent state. The existing reloading schemes can be divided into two classes: *simple reloading* and *concurrent reloading*. In simple reloading, the system does not resume its normal operation until the entire database is brought back into the main memory. In concurrent reloading, reload activities and transaction processing are performed in parallel. More specifically, the database is reloaded on demand on a per-partition basis [3].

3. Implementation

We have chosen to implement, with some adjustments, the recovery algorithm as outlined in [1]. Our MMDB recovery system implements physical logging, non-fuzzy, action consistent checkpointing and simple reloading. The key features of the algorithm are:

- The write-ahead log on disk contains only the redo records of committed transactions, minimizing recovery I/O. The redo and undo logs are maintained in main memory for active transactions. Once a transaction is committed, the undo log is discarded. When a transaction aborts the undo and the redo logs are discarded. The undo log is written to disk only at the time of checkpointing, while the transaction is active. The redo log is forced to disk when the transaction commits.
- During recovery, only a single pass is made over the logs on disk. The usual backward pass on the logs to find the ‘winners’ and ‘losers’ and to undo the action of the losers is avoided by keeping separate undo and redo logs [1].
- The technique can be used with physical as well as logical logging.
- Since checkpointing is done in an action consistent manner, interference during normal transaction processing is kept to a minimum.
- No assumptions are made regarding the availability of special hardware such as non-volatile RAM or an adjunct processor for checkpointing [1]. The scheme can be used with any standard machine configuration.

In the following sections, we describe the system model used and delineate the specific aspects of the recovery mechanism that we implement.

3.1 System Model

We assume that the entire database is kept in main memory while a backup copy, possibly out of date, is kept on disk. We assume that the disk storage is stable and will never lose its content. The system maintains a redo log on disk. Information about actions that update the database is written to the redo log

so that the actions can be redone at restart time after a crash. A private redo and undo log is maintained in main memory for each active transaction and the redo log is forced to stable storage when the transaction commits. The portion of the redo log on disk is referred to as the *persistent redo log* and the portion of the redo log in main memory as *volatile redo log*. The entire redo log is referred to as the *global redo log*.

Figure 1 shows the general structure of a log record and **Figure 2** shows the system model.

Trans. No.	Log Type	Data
------------	----------	------

Trans. No.: *The ID of the transaction writing the log record*

Log Type: *Begin transaction, Operation (e.g. update), Commit transaction, Abort transaction*

Data: *Tuple data*

Figure 1. Log Record

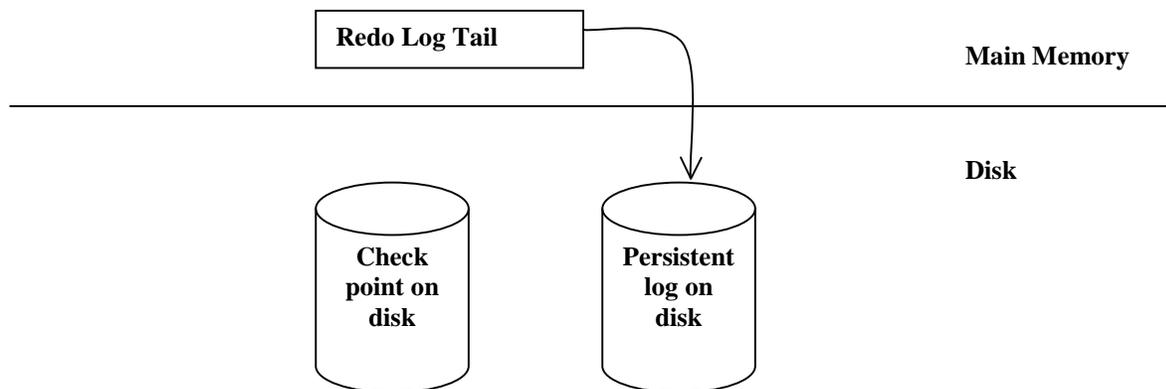
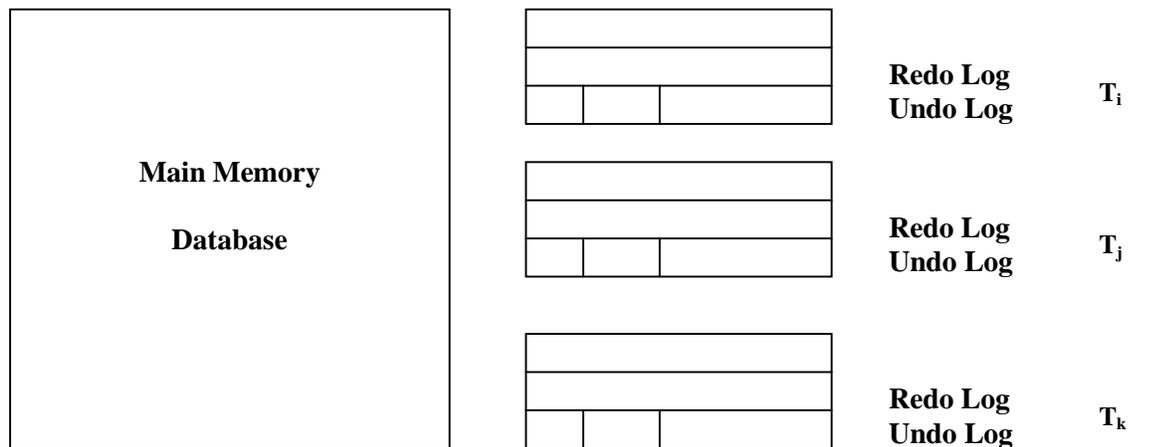


Figure 2. System Model

3.2 Data Structures

The system maintains on disk the *global redo log*. Only a committed transaction may reside in the global log, where all of its redo actions appear consecutively. To achieve this, the redo records of an active transaction are initially kept in a private redo log in the main memory and are subsequently appended to the global log at the commit time of the transaction. The undo records of an active transaction are also kept in the main memory in a private log, separate from the redo log of the transaction. Unlike the redo records, the undo log records are written to disk only when a checkpoint takes place. The database we maintain is essentially an array with each element of the array (*tuple*) a structure of integer fields. The size of each tuple (*i.e. the number of fields in the tuple*) can be varied by changing a constant. Each field is of type *integer*, which is **4 bytes** long on the architecture of the machine on which we run the database system.

We maintain physical log records for both the undo and redo logs. Each log record consists of four fields. The transaction ID field identifies the transaction that effects the update. The type field identifies the type of the log record, which can be of type “**begin**”, “**op**”, “**commit**” or “**abort**”. The index field (*array index*) identifies the physical location of the modified data. The data field contains the “**before-image**” (*in the case of an undo record*) or the “**after-image**” (*in the case of a redo record*) of the modified data. We also maintain a *global list of active transactions* as a **linked list**. Whenever a transaction starts, the transaction number is inserted into the list. When the transaction commits or aborts, the corresponding transaction number is removed from the linked list.

3.3 Normal Processing

When a transaction T_i starts its execution, it obtains all the necessary locks for its updates. It is then added to the list of active transactions, and the record $\langle \text{begin } T_i \rangle$ is written to the private redo log of T_i . During the lifetime of T_i , its undo and redo records are maintained in its private logs.

To prevent deadlocks, only one transaction may acquire locks for updates at any time. In addition, to ensure that a checkpoint takes place in an action consistent state of the database, we maintain an action-consistency lock that covers the entire database. Any update action has to acquire the lock in shared mode before the start of the action and release it after the action is complete. However, the checkpointer acquires the action lock in exclusive mode before the start of the checkpoint and releases it after the checkpoint is complete (refer to **Section 4**).

3.4 Commit Processing

When a transaction T_i is ready to commit, the database system takes the following actions:

- Write a $\langle \text{commit } T_i \rangle$ to private redo log of the transaction
- Append the private redo log of the transaction to the global log on disk
- Remove the transaction from the list of active transactions
- The transaction releases all the locks that it holds

Note that only one transaction is allowed to commit at any time.

3.5 Abort Processing

When a transaction T_i aborts, the database system takes the following actions:

- Performs all the undo operations of the transaction by traversing its private undo log backwards
- Writes $\langle \text{abort } T_i \rangle$ to the private redo log of the transaction
- Appends the private redo log to the global log
- Removes the transaction from the list of active transaction
- The transaction releases all the locks that it holds

4. Checkpointing and Recovery

In this section, we describe the details of checkpointing and of recovery.

4.1 Checkpointing

Checkpointing is done in an action consistent manner (*i.e. no update actions are in progress during checkpointing*). When the checkpointer is invoked, it does the following actions:

- Freezes all access to the database by acquiring the action-consistency lock in exclusive mode
- Write the following to the new checkpoint image on disk:
 - the main-memory database
 - the pointer to the end of the global redo log
 - the list of active transactions
 - the undo logs of all active transactions
- Remove the old checkpoint image

Note that the above information does not overwrite the information taken during the previous checkpoint. The old information is deleted only after the current checkpoint is complete. Thus, we ignore any partially written checkpoint in the event of a crash and use the most recent *complete* checkpoint image for recovery.

Thus, this scheme ensures that:

1. The undo log record is on stable storage 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 storage before a transaction is allowed to commit, so that its updates can be redone if necessary

4.2 Reloading

The reloading algorithm implemented makes a single pass over the persistent log. Reloading is executed on restart after a system crash, before the transaction processing is allowed (*simple reloading*).

The recovery system does the following during reloading:

- From the most recent complete checkpoint, read into the main memory
 - the database
 - the pointer to the end of the persistent global log at checkpoint time
 - the list of active transactions at checkpoint time
 - the undo logs of all transactions active at checkpoint time
- Starting from the end of the persistent log end noted in the checkpoint and scanning forward the redo log, redo all redo operations on the database
- Perform the undo operations, using the checkpointed undo log for those transactions that are in the checkpointed list of active transactions, *and:*
either
 - whose abort records **are found** in the global redo log,*or*
 - whose commit records **are not found** in the global redo log
- After all the undo operations of a transaction are performed, an abort record for the transaction is written to the global log

All redo records of transactions that do not have a corresponding commit record need to be ignored. So while reading the redo log, we read in all the records for a transaction (*which are in consecutive order*), instead of traversing the log backwards to skip them, before performing any action. If the commit or abort record for the transaction is not found, we ignore the log records of the transaction, which were read in earlier.

5. Testing and Correctness

Our database is defined to be an array of *DB_SIZE* tuples, each element of which is itself an array of *NO_FIELDS* integers. Each field in the tuple is 4 bytes long. In this way, we can simulate relations consisting of any number of tuples and any number of fields in each tuple.

In our implementation, the main memory database system uses UNIX threads (*POSIX*) to represent transactions. Each thread starts a transaction, does the transaction processing, commits or aborts the transaction and then starts a new transaction. All but two threads (*#_THREADS*) are devoted to transaction processing, with each representing a single transaction at any instant of time. Each transaction is of one of two types: either *updating the database* or simply *reading tuples* from the database. The ratio of transactions performing these actions is tunable.

We maintain a global transaction counter and assign a unique transaction identifier to each transaction created by a thread. The granularity of the locking is at the tuple level. We provide the **standard 2 phase locking protocol** for all transactions. Each transaction must acquire locks on the tuples it wants to access before it can start processing.

Each transaction is made up of one or more actions. The number of actions that each transaction gets is defined at transaction creation by a random number generator. We define the maximum number of actions per transaction. The random number generator generates numbers between one and that maximum and assigns the number of actions to each transaction.

Each update transaction also has a *think time*, which is an input parameter of the program. Of the two non-transaction threads, one emulates the checkpointing, activated every *n* seconds, where *n* is also an input parameter. The other non-transaction thread simulates the system crash. The crash thread is put to sleep for an amount of time, which is another input parameter (this time indicates the total run time of the program before the system crashes), before forcing an exit, thus simulating a crash that may happen at any time.

To test the correctness of our recovery scheme, the crash thread creates a database image that must be matched by the recovery process, before forcing the exit. It takes a snapshot of the database, performs, for all active transactions, the necessary undo operations on the snapshot and saves the snapshot to disk. At system restart, the recovery system performs the database recovery, restoring the database to a consistent state using the information in the most recent complete checkpoint and the redo and undo logs. We then compare the restored database to the snapshot image taken at the crash point. (One way of accomplishing this is to use the UNIX *cmp* command on the snapshot and the restored database.) The comparison showed that the restored database matches the snapshot of the database at crash time, thereby validating the correctness of our recovery implementation.

6. Performance

There are two dimensions in our study of the performance of the current implementation. First, we examine the performance of the recovery process (*recovery time*) as a function of the number of tuples in the database (*Figure 3*), for two-, five- and ten-field tuples respectively, where the size of each field is 4 bytes. Second, we fix the size of the database (*constant memory*) and vary the ratio between the total number of tuples and the number of fields in each tuple (*Figure 4*). This is done for 8MB, 16MB and 32MB databases respectively. This indicates the influence of the number of tuples in the database as compared to the number of fields in each tuple (keeping the total database size constant) on the recovery time. We see that reducing the number of tuples in the database significantly reduces the recovery time even though the number of fields in a tuple is increased.

Figure 3: Recovery Time

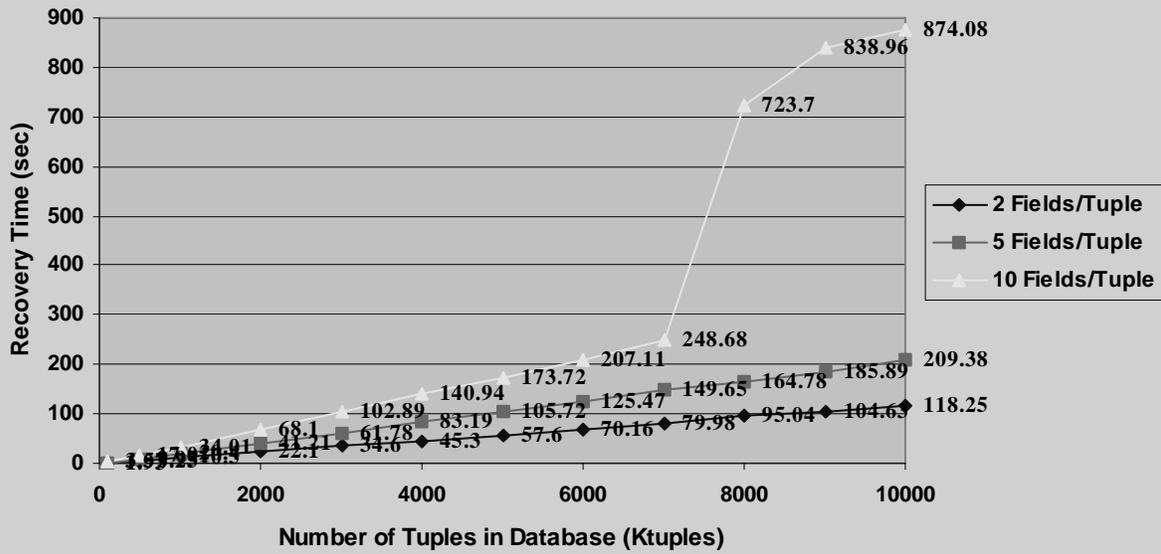
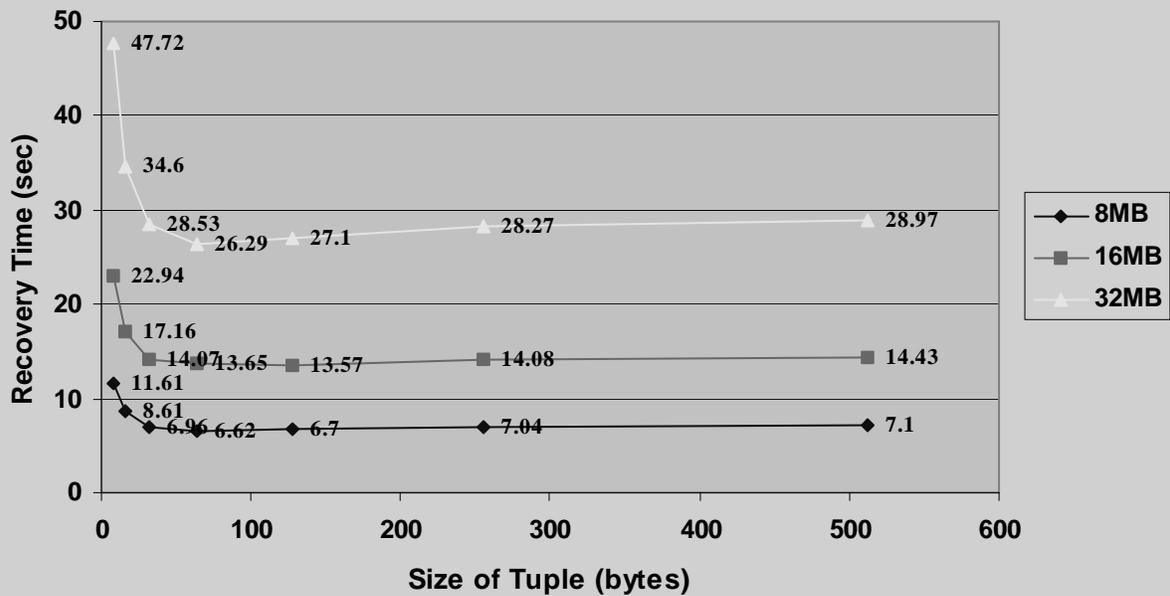


Figure 4: Recovery Time for Constant-Sized Database



6.1 Discussion of Graphs

Figure 3 shows the recovery time as a function of the number of tuples in the database. On the x-axis we plot the number of tuples in the database (**in Ktuples**), which ranges from 100K tuples to 10M tuples. The recovery time **in seconds** is plotted on the y-axis. We also vary the number of fields in a tuple to plot three different curves. The numbers of fields per tuple plotted are 2, 5 and 10.

We see from the graph that the cost of recovery (*in terms of recovery time*) grows linearly as the number of tuples in the database increases. It also shows, more subtly, that the cost growth is decreasingly sensitive to the widening of the tuples. For example, take the performance figures for 4000K tuples. The cost ratio between five- and two-field tuples is 1.84, whereas the cost ratio between ten- and five-field tuples is 1.69.

For the case of the 10-field-per-tuple curve, we see a sharp increase in the recovery time when we increase the number of tuples in the database from 7MB to 8MB. We were initially quite surprised to see this jump in the recovery time. We later found that the increase was attributed to the fact that the total size of main memory consumed by our program (*not the size of the database*) exceeded the available memory and that as a result, the virtual memory swapping mechanism swapped our program to disk.

Figure 4 shows the recovery time for a constant size database, varying the ratio between the number of tuples in the database and the number of fields in the tuple, keeping the total size of the database constant. On the x-axis we plot the size of a tuple (**in bytes**) and on the y-axis we plot the recovery time (**in seconds**). The three curves correspond to the database sizes of 8MB, 16MB and 32MB respectively. As the size of the tuple increases (with the number of tuples decreasing to maintain a constant size of the database), the cost decreases dramatically. The recovery time only increases gradually beyond the optimal point (when the size of the tuples is substantial compared to the number of tuples in the database). This graph also proves the fact presented earlier that the recovery time is very sensitive to the number of tuples in the database compared to the size of a tuple.

7. Future Direction

Our implementation of the main-memory database uses non-fuzzy checkpointing currently. Before the checkpointer can do checkpointing, it gets an action lock, and as a result, during checkpointing all transactions are halted. An obvious improvement is to use fuzzy checkpointing algorithms, which reduce the interference of the checkpointing process with normal transaction processing. Moreover, instead of dumping the whole database to the checkpoint file, the database may be dumped in segments to allow some transactions to proceed during checkpointing. Checkpointing time can also be reduced by copying only the modified pages of the database to disk instead of the whole database. Finally, it would be interesting to see the performance of the recovery system if we implement concurrent reloading instead of the simple reloading mechanism.

8. Conclusion

As main memory gets cheaper and hundreds of megabytes of memory become readily available, main-memory databases will prove to be conducive to applications that require high performance and very fast response times. The only disk I/O that is required by such a database arises from the recovery sub-system. In this project, we have implemented a recovery system for a main-memory database and have studied the performance of the recovery mechanism. We find that the recovery time increases linearly as the number of tuples in the database increases and that the recovery time is very sensitive to the number of tuples in the database compared to the size of each tuple in the database.

Acknowledgement

We would like to thank **Prof. David DeWitt** for his guidance and useful suggestions throughout the life span of this project. It was always encouraging and motivating to see his smiling face and his eagerness to help us, even when we barged into his room without an appointment.

References

- [1] H.V. Jagadish, Avi Silberschatz, S. Sudarshan, “*Recovering from Main-Memory Lapses*”, Proceedings of the 19th VLDB Conference Dublin, Ireland 1993
- [2] Le Gruenwald, Margaret H. Eich, “*MMDB Reload Algorithms*”, Proceedings of the ACM SIGMOD International Conference on Management of Data, 1991, pp. 397-405
- [3] Le Gruenwald Le, Jing Huang, Margaret H. Dunham, Jun-Lin Lin, Ashley Chaffin Peltier, “*Survey of Recovery in Main Memory Databases*”, Engineering Intelligent Systems 4/3, September 1996, pp. 177-184
- [4] Kenneth Salem, Hector Garcia-Molina, “*Checkpointing Memory-Resident Databases*”, Proceedings of IEEE CS International Conference No. 5 on Data Engineering, Los Angeles, February 1989, pp. 452-462
- [5] Jun-Lin Lin and Margaret H. Dunham, “*Segmented Fuzzy Checkpointing for Main Memory Databases*”, Proceedings of the ACM Symposium on Applied Computing, February 1996, pp. 158-165
- [6] Margaret H. Dunham, Jun-Lin Lin, and Xi Li, “*Fuzzy Checkpointing Alternatives for Main Memory Databases*”, Recovery Mechanisms in Database Systems, edited by Vijay Kumar and Meichun Hsu, Prentice-Hall Inc., pp. 574-616
- [7] David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael Stonebraker, David A. Wood, “*Implementation Techniques for Main Memory Database Systems*”, SIGMOD Conference 1984, pp. 1-8
- [8] Margaret H. Eich, “*Main Memory Databases: Current and Future Research Issues*”, IEEE Transactions on Knowledge and Database Engineering, Vol. 4, No. 6, December 1992, pp. 507-508 (Introduction to Special Section on Main Memory Databases)
- [9] Bohannon, P., D. Lieuwen, R. Rastogi, A. Silberschatz, and S. Sudarshan, “*The Architecture of the Dali Main Memory Storage Manager*”, The International Journal on Multimedia Tools and Applications 4, 2, March 1997
- [10] Rastogi, R., P. Bohannon, J. Parker, R. Rastogi, S. Seshadri, A. Silberschatz, and S. Sudarshan, “*Distributed Multi-Level Recovery in Main-Memory Databases*”, Distributed and Parallel Databases 6, 1, January 1998
- [11] P. Bohannon, D. Lieuwen, A. Silberschatz, S. Sudarshan, and J. Gava, “*Recoverable User-Level Mutual Exclusion*”, Proc. IEEE Symposium on Parallel and Distributed Computing, October 1995
- [12] Rajeev Rastogi, S. Seshadri, Philip Bohannon, Dennis W. Leinbaugh, Abraham Silberschatz, S. Sudarshan, “*Logical and Physical Versioning in Main Memory Databases*”, VLDB 1997, pp. 86-95