Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

C.A. van den Berg, M.L. Kersten

Logging and recovery in PRISMA

Computer Science/Department of Algorithmics & Architecture Report CS-R9049 September

Logging and Recovery in PRISMA

C.A. van den Berg
M.L. Kersten

Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam
The Netherlands

Abstract

The recovery methods for main-memory database systems are mostly based on maintaining a disk based log and checkpoint. The information written to the log usually consists of low level information, before-after images or single tuple updates. In this paper we examine the effect of writing higher level update information to the log. This results in less logging overhead during normal processing, but an increased replay overhead during recovery.

In this paper we introduce a cost model for analyzing the effect of higher level logging on the transaction throughput, and present the results of a performance analysis based on this cost model.

Key Words & Phrases: main memory database machines, logging and recovery, performance analysis.
1985 Mathematics Subject Classification: 69H22, 69H26, 69C40, 69C24.
1990 CR Categories: H.2.2, H.2.6, C.2.4, C.4.

1 Introduction

An advantage of main-memory DBMS over disk-based systems is the performance gain that results from ignoring one level in the storage hierarchy during retrieval. However, to ensure stability either its memory should be non-volatile, i.e. safe RAM, or disks should be used to keep logs and checkpoints. PRISMA [1] follows the latter approach; using multiple disks to realize a stable storage subsystem.

The overhead involved in maintaining a log and checkpoint pair has been identified as a limiting factor for the transaction throughput. Traditionally logging takes place at the physical record level, which reduces the time spent in reconstructing the actual database from the most recent checkpoint. Logging at a logical level, for instance of SQL update statements, can greatly reduce the amount of log information stored, and therefore reduce the IO cost involved in reading the log. The price being paid is an increased CPU cost, due

[1]The PRISMA project is a SPIN project, partially funded by the Dutch government, and is a joint effort of Philips Research Labs, the CWI and several Dutch universities for the development of a parallel inference and storage machine.

Report CS-R9049
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

to longer log replay. This cost may be reduced considerably by the exploitation of parallel hardware. In general a tradeoff must be made, between the overhead of logging during normal processing and the time spent in crash recovery.

Several techniques are proposed to reduce the overhead for logging and checkpointing , including parallel logging [AD85], using stable log storage [DKO$^+$84, Eic87], and database partitioning [LC87]. It is for memory resident databases not entirely obvious at which level the logging process should take place. In this paper we present a cost model, which is used to examine the effect of the logging level. This model is also used to study the effect of parallel logging, and database partitioning on the transaction throughput in the context of the PRISMA architecture.

The remainder of this paper is organized as follows. Section two contains a short introduction to the PRISMA architecture and gives an indication of related work. In the third section the PRISMA recovery architecture is presented. A cost model for this architecture can be found in section four and, finally, we present the results of some experiments based on this cost model.

2 Background and scope

2.1 PRISMA and POOL

The PRISMA database management system is developed fo a prototype multiprocessor machine, which consists of 100 processing nodes interconnected through a dedicated packet switching network. Each processing node is based on a MC68020 with 16 Mbyte of non-stable memory. Every two nodes share a 150Mbyte disk (See elsewhere in this volume).

The PRISMA database system is implemented in POOL-X [Ame88] a parallel object oriented language designed to simplify the construction of parallel applications. The language provides a stable storage class for data persistency. This class offers the same operations as an ordinary typed file (à la Pascal), but the implementation ensures that writes are atomic and that they survive system crashes and disk failures.

The language does not provide a low level interface to disk, which would allow reading and writing pages directly and fancy disk allocation algorithms for minimizing average seek time. Furthermore, the DBMS designer has no control over the physical records, nor direct access to the system buffers to speedup checkpointing. Everything below the simple file interface is hidden. Consequently logging and checkpointing can only be implemented in terms of POOL objects.

2.2 Related work

In [LC87] Carey and Lehman propose to perform the log and checkpointing operation on database partitions. A partition is used to store a variety of information, ranging from indices to tuples. The logfile and checkpoint file for a partition contains enough information to allow for an independent recovery of the partition. Using demand recovery improves the average response time of transactions further. The undo records for update transactions are stored in volatile memory. A separate recovery processor is used to write the redo log entries of committed transactions to stable storage.

In PRISMA the database relations are already partitioned into fragments. In many respects, the fragment storage can be seen as a partition in terms of [LC87] and, thus, could be the level the recovery and logging process.

In [DKO$^+$84] DeWitt compares several recovery schemes for disk-resident multi-processor database machines. Methods based on logging, shadowing, and differential files are compared. The experiments show that a recovery scheme based on parallel logging is most suited for multiprocessor database machines. In particular it proves to be an effective technique for improving the transaction throughput. Although PRISMA is foremost a main memory machine, the observation of DeWitt is still relevant, because the overhead involved in gathering the recovery information is considered.

The paper by Eich [Eic87] gives a classification of recovery architectures for main memory database systems. The different classes are compared on transaction throughput and transaction cost (response time). The classification is based on: (1) the amount of stable memory (none, only log, and all), (2) availability of logging hardware (yes, no), (3) checkpointing overhead (yes, no) and (4) commit policy (immediate, group and precommit). From a simple analytical model the following general conclusions are drawn: (1) group commit is bad for response time for an individual transaction, (2) stable memory and log processor is good for both response time and throughput, (3) if there is no stable memory, use group commit to improve throughput. The cost model presented, ignores the effect of the level level at which the logging takes place.

In the remainder, we investigate the influence of the logging level on the transaction cost and throughput. Our primary goal is to find a recovery architecture suitable for the PRISMA machine. Thus, we have to ignore the evidently good approaches to equip the system with some safe RAM [CKKS89]. The software techniques like partitioning and parallel logging can be used, however.

The next section presents the considerations for the design of the PRISMA recovery architecture.

3 Recovery architecture

In this section we describe the general architecture of PRISMA and describe how the ideas of parallel logging, database partitioning, and choosing a different logging level work out in the PRISMA context.

We start with a brief description of query processing within PRISMA. A more detailed description of the architecture can be found in [KAH$^+$87]. Then we define the causes for transaction failures and, finally, we present three alternative logging techniques for PRISMA, which are based on varying the level at which logging takes place.

3.1 PRISMA query processing

The PRISMA database system is designed as a distributed relational database system. The database relations are fragmented and stored in data managers, called One Fragment Managers (OFM). A fragmentation rule tells how the relation can be reconstructed from its fragments. Currently, horizontal fragmentation is supported only. Following the fragmentation rules a query on a relation is translated to queries on its composing OFM's, which can be executed in parallel.

Transactions are translated by an SQL parser to its equivalent form in XRA, which stands for eXtended Relational Algebra. A description of this language can be found in [WG89]. Apart from the ordinary set of relational operators like select, join, group an project, XRA offers operators for handling recursive queries and operators to express and control parallel execution of subqueries.

The Query Optimizer transforms the XRA statements to a semantically equivalent but less costly set of XRA statements. The optimizer uses the fragmentation rules to expres the query in terms of operations on fragments. In the sequel we refer to the two forms of the XRA query as XRA-R (on relations) and XRA-F (on fragments).

The Transaction Manager takes the XRA-F transaction, requests locks for the involved fragments and passes the individual OFM's the required XRA-F statements. Before a transaction commits, the Transaction Manager checks the integrity constraints, which must hold for the database relations. Transaction atomicity is ensured using a standard two phase commit protocol.

The individual OFM's execute the XRA-F statements on their fragment data. Each XRA-F update operation, which is set oriented, can result in a sequence of tuple insertions and deletions.

3.2 PRISMA transaction failures

The recovery mechanism of a database system must be able to recover from different causes for transaction abort. Following the overview on transaction oriented recovery given in [HR83], we distinguish between transaction failures, system failures and media failures.

Transaction failures are likely to happen frequently, about 1% - 10% of all update transactions [Reu84]. The recovery from these failures should therefore be fast. In PRISMA, like other main memory database systems, we keep the *undo* logrecords in volatile memory, which results in a fast recovery from these failures.

System failures are caused by hardware failures, operating system failures, and database system software failures. Estimations based on the failure rate of the hardware components in PRISMA, indicate that a hardware caused system crash occurs every three days [Mul87]. For the other causes of system failure we can not give a reliable estimation. The machine recovers from a system crash by performing a cold restart. Thus in the event that one processing node fails, the whole system is rebooted. [2]

[2]The problem of building a fault tolerant system was considered to be out of the scope of this project.

The database is protected against media failures by using replicated files as a backup storage for the fragment data. The replicas are allocated by the operating system to different disks. After a system crash, the operating system restores the file system into a consistent state. Thus recovery from media failures is performed by the operating system.

3.3 PRISMA logging and recovery

A transaction is any sequence of XRA (update) statements bracketed as such. The Transaction Manager is in charge of guarding the transaction semantics, i.e. atomicity, consistency, isolation and durability. The recovery mechanism assures that the effects of committed transactions survive system crashes.

The recovery mechanism for PRISMA maintains a log file, where the update statements for a transaction are recorded. The logfile is also used to record the transaction status information. The transaction status can be aborted or committed. In the event of a system crash, the updates of all committed transactions, which are recorded on the logfile are replayed. To reduce the replay time, the database is checkpointed once the log size reaches a certain threshold value. A low treshold value incurs too much checkpointing overhead during normal processing. However, if the treshold value is chosen too large, log replay during recovery becomes too time consuming [CBDU75]. Clearly, we need a threshold value for which the overhead is minimal.

In PRISMA a two phase commit protocol is used to obtain transaction atomicity [CP84]. For the design of the recovery mechanism, which is integrated in the two phase commit protocol, two choices have to be made:

- Which processes are involved in logging ?
- What information is logged ?

The logging process can either be centralized or distributed. In centralized logging the Transaction Manager could log all the update information. In distributed logging, both the OFM's and the Transaction Manager could be involved in the logging process. For distributed logging the Transaction Manager records the global abort or commit decision on the transaction log only. The OFM's record the updates performed on behalf of the transaction, the local abort or commit decision, and the global decision.

The design of the logging and recovery scheme for PRISMA aims for the best possible transaction throughput. That is, we do not optimize the logging procedure or the recovery procedure in isolation, but together in relation to the expected mean time between failure (MTBF). The choice of what is logged has a direct influence on the crash recovery time and the logging overhead. Evidently, logging at the highest level in PRISMA leads to small amounts of log info. But the cost associated with recovery could become so high that the effective transaction throughput remains low. So there is a tradeoff between IO and CPU cost.

In the following overview we consider, the influence of a given log level on the replay cost and logging cost for a transaction. Three different logging levels are considered: XRA-R, XRA-F and Tuple level. We illustrate the effect using the following update transaction:

$$\begin{array}{ll} UPDATE & R \\ SET & salary = salary * 1.10 \\ WHERE & dept = \text{``}CS'' \end{array}$$

3.3.1 XRA-R

The SQL transaction is translated by the SQL parser to the following XRA-R transaction:

$$\begin{array}{l} update(R, select(R, dept ='' CS''), salary = salary * 1.10) \\ commit/abort \end{array}$$

The logging process is controlled by the Transaction Manager, which writes the transaction commit or transaction abort decision to the log. A checkpoint operation is initiated by the Transaction Manager, whenever the log size reaches a certain threshold. Update operations are only written to the log if all the locks for the operation have been acquired. This ensures that during replay the transactions are reexecuted in the correct order.

The recovery process requires that all OFM's reload their latest checkpoint and that the Transaction Manager, in charge of the recovery, replays the operations of committed transactions found on the log. The replay cost includes the translation of the XRA-R statements to XRA-F statements.

3.3.2 XRA-F

Logging at the XRA-F level is just like logging at the XRA-R level a form of centralized logging. The advantage is that the XRA-F statements are already optimized. The amount of log information., however, is larger. The single example XRA-R statement is translated to n_f XRA-F statements, if the relation R is stored in n_f fragments. [3]

$$\begin{array}{l} update(F_i, select(F_i, dept ='' CS''), salary = salary * 1.10) \quad i = 1, \ldots, n_f \\ commit/abort \end{array}$$

The Transaction Manager records the XRA-F statements and the transaction commit and transaction abort decisions on the log. The checkpoint operation is again initiated by the Transaction Manager.

The recovery process involves reloading the latest checkpoint in the OFM's and replaying the XRA-F log. [4]

[3]This number of logrecords is a pessimistic value and probably too high in practice.

[4]It is possible with this scheme to recover fragments on demand by analyzing the dependency graph for fragments and transactions. This analysis delivers a subset of the transactions on the transaction log, which is minimally required to recover the database to a correct state. It general, it is unlikely that this subset is smaller than the complete collection of completed transactions on the transaction log.

3.3.3 Tuple

Logging at the tuple level is a form of distributed logging. It is basically the technique described in [CP84]. Each XRA-F statement results in a list of tuple updates. The exact amount depends on the selectivity of the update and the size of the fragment s_f. In PRISMA a hash-based fragmentation scheme is used, thus each fragment has on the average the same amount of tuple updates.

$$\begin{array}{ll} update(tuple_i, f(tuple_i)) & i = 1, \ldots, \sigma s_f \\ precommit & \\ commit/abort & \end{array}$$

The OFM's write the list of tuple updates to a private log. If the size of the log becomes too large, a checkpoint of the fragment data is made. Because the One Fragment Manager has complete control over the updates on the fragment it is also possible to perform a 'fuzzy' checkpoint operation. By keeping a copy of the data of completed transactions, the checkpoints can be written to disk, while new update transactions are in progress. This process is described in a little more detail in [DKO+84].

For recovery it is necessary that the OFM records both the precommit, as well as the global commit or global abort records for the two phase commit protocol. The Transaction Manager writes the global decision on a system log, because a crash may occur before the global decision is recorded on the logs of all OFM's involved.

The recovery of an OFM then involves reloading its most recent checkpoint and replaying the updates of completed transactions from its private logfile. Transactions, which have entered their precommit phase, but which have not yet run to completion are either aborted or committed, depending on the information on the system log.

Because the logging and checkpointing information involves only a single fragment, it is possible to recover fragments individually. This makes on demand recovery of fragments possible.

4 Cost model

In the previous section, we have discussed in general terms the recovery mechanisms for each logging level and argued that there is a tradeoff between logging overhead during normal processing and recovery time. In this section, we want to substantiate this claim by deriving a simple cost model for a memory resident database system. Our model is based on the one proposed by Eich [Eic87] for main memory database systems to experiment with different logging policies. The main difference is that we model the recovery time explicitly, as it influences the throughput.

The cost model is based on a simplified PRISMA architecture consisting of several processors connected by a communication network and sharing a single disk (see figure 1). The effect of using several disks in parallel can be modeled by reducing the time to read or write to disk. The fragments of a relation are allocated to different processors and we

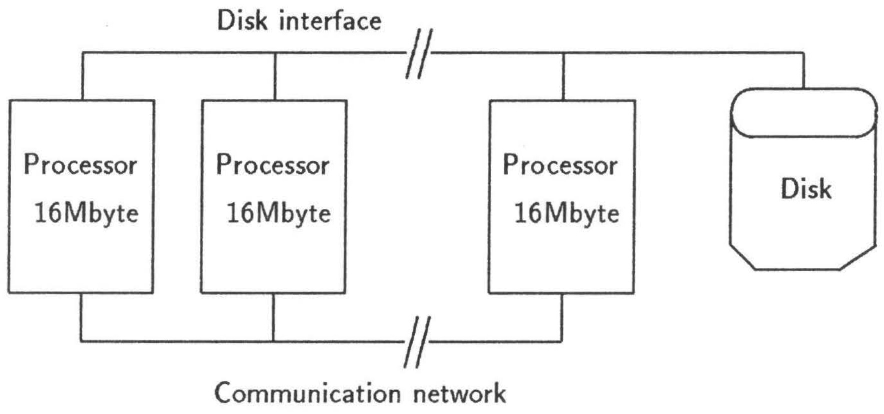


Figure 1: The architecture for the cost model

assume that an update transaction affects a single relation only. This means that the tuple modifications, during normal processing and log replay can be performed in parallel for each fragment.

The procedure for determining the proper logging level is as follows. First, we express the transaction throughput as a function of the checkpointing frequency. Next, we determine the checkpointing scheme that maximizes transaction throughput. Finally, this cost formula is used to determine the maximum transaction throughput for different logging levels, using some typical cost estimates for the basic operations, like writing log records and copying memory.

4.1 The basic cost model

A summary of the parameter setting is given in table 1. In the remainder σ represents the update selectivity; that is the fraction of the relation, modified by an update transaction.

The transaction throughput r_t depends on the MTBF T_f, the recovery time c_R and the average transaction time c_t. We assume that the recovery must be finished before normal processing can begin.

$$r_t = \frac{T_f - c_R}{c_t T_f}$$

The recovery cost is determined by the cost for reloading the latest database checkpoint and the cost for replaying the log records. The reload cost depends on the database size s_{db} (in pages) and the time to read a page from disk c_{io}.

The checkpointing policy for all logging levels considered is that L_{max} log records are written, after which a new checkpoint is produced. The replay cost consist of the IO cost for reading a log record c_l and of the CPU cost for re-executing the log record c_{replay}. Obviously c_{replay} depends on the logging level.

$$c_R = s_{db} c_{io} + \frac{L_{max}}{2}(c_l + c_{replay})$$

$$s_{db} = n_r n_f n_t s_t$$

SQL	0.6	SQL level execution overhead
$XRA-R$	0.4	XRA-R level replay overhead
$XRA-F$	0.2	XRA-F level replay overhead
$Tuple$	0.0	Tuple level replay overhead
f_u	0.25	the fraction of update transactions
f_t	0.03	the fraction of failing update transactions
s_l	0.1	the size of a log record in pages
s_t	0.5	the size of a tuple in pages
n_t	1000	the number of tuples in a fragment
n_f	10	the number of fragments in a relation
n_r	30	the number of relations in the database
T_f	259200	MTBF
c_{io}	0.03	time to read/write a page to/from disk
c_{qry}	0.8	average cpu time per read transaction
c_{tuple}	0.001	tuple update cost

Table 1: The parameter setting

The average transaction time c_t is determined by assuming that a fraction f_u of all transactions are update transactions. We assume that the cpu cost involved in read transactions can be estimated by constant cost c_{qry}. The effect of parallel execution is already included.

$$c_t = (1 - f_u)c_r + f_u c_u$$

$$c_r = c_{qry}$$

The cost for an update transaction depends on the cost for updating the data c_{upd}, the number of log records produced by the transaction n_l, on the proportional overhead required for the checkpoint operation c_{chk}, the checkpoint frequency f_c and the fraction of transaction failures f_t. Note that the number of log records produced in a transaction depends on the log level.

$$c_u = c_{upd} + n_l c_l + f_c c_{chk} + f_t c_{undo}$$

The update cost consists of a fixed amount for translating the SQL update operation to tuple updates and on a variable cost for modifying the selected tuples. The latter depends on the update selectivity σ and the fragment size n_t.

$$c_{upd} = SQL + \sigma n_t c_{tuple}$$

Every time the total number of log records written to the log exceeds the threshold value L_{max}, a checkpoint is generated. This results in a fraction of $\frac{n_l}{L_{max}}$ checkpoint operations per update transaction.

The time spent in the checkpoint operation depends on the amount of dirty pages produced during normal processing. We make the assumption that the (tuple) updates are equally distributed over all the database pages. Using probabilistic arguments we find an expression for the expected amount of dirty pages after k tuple updates.

Between two checkpoints run $\frac{L_{max}}{n_l}$ update transactions, which produce together $k = \sigma n_f n_t \frac{L_{max}}{n_l}$ tuple updates.

$$c_{chk} = (1 - (1 - \frac{1}{s_{db}})^k)) s_{db} c_{io}$$

The cost for undoing the result of a transaction is determined by the amount of updates already performed on a fragment. As the list of undo records is kept in memory, no IO cost is involved. Therefore, the update selectivity σ determines the amount of tuple modifications, which have to be undone.

$$c_{undo} = \sigma n_t c_{tuple}$$

Given the formula for the transaction throughput can determine the maximum throughput for each logging level by filling in the n_l and c_{replay} parameters and differentiating the transaction throughput.

$$\frac{d}{dL_{max}} r_t = 0$$

As the solution of this equality is analytically intractable, we have solved the equality numerically.

4.2 Analysis of the logging levels

The cost formula $r_t(L_{max})$ is parameterized by the replay cost c_{replay} per log record and the number of log records per transaction n_l.

The replay cost depends on the number of transactions, which can be replayed and the cost per transaction. On the average about $\frac{L_{max}}{2n_l}$ update transactions have to be replayed after a system crash. This is per log record $\frac{1}{n_l}$.

The transaction cost for each logging level depends on a level dependent overhead c_{level} and replay cost for the tuple updates, which depends on the update selectivity. The replay of the tuple updates can be performed in parallel for each fragment and therefore only depends on the size of the fragment.

$$c_{replay} = \frac{c_{level} + \sigma n_t c_{tuple}}{n_l}$$

For the XRA-R level 2 log records are produced per transaction. The replay cost per log record involves the translation of the XRA update statement and the subquery execution in the OFM, which are performed in parallel.

The XRA-F level produces $1 + n_f$ log records. The replay cost per log record is again composed of a fixed overhead for translating the XRA-F statements and the actual tuple updates.

	XRA-R	XRA-F	Tuple
n_l	2	$1 + n_f$	$1 + 2n_f + \sigma n_f n_t$
c_{level}	$XRA - R$	$XRA - F$	0
logging method	centralized	centralized	distributed

Table 2: Parameter setting for different logging levels

At the tuple level for each tuple update a log record is produced. The replay cost per log record is simply the cost for performing a single tuple update. As the log records for the n_f fragments can be replayed in parallel. An overview of the parameter settings for the different logging levels can be found in table 2.

5 Experiments and results

The cost model presented formed the basis for some experiments to increase our understanding of the parameter settings. These experiments show that choosing a higher log level as the basis of the recovery architecture improves the transaction throughput of the database system. Additional experiments were conducted to validate that this conclusion holds even when the parameters for the hardware change an order of magnitude, and if the ratio read/update transaction shifts. All calculations are based on the default parameter settings, which can be found in table 1.

Critical in optimizing the recovery mechanism is the choice of the threshold log value. This value is different for each log level, because it is determined by the replay cost and the log cost. For tuple level logging the maximum transaction throughput is reached at much higher values for the log size threshold, than for XRA-R and XRA-F level logging. This is caused by the checkpointing overhead. For tuple level logging, the threshold value L_{max} is reached sooner than for XRA-R and XRA-F level logging, which results in more checkpointing overhead per transaction. An increase in the update selectivity necessarily results in a reduced transaction throughput, but has no effect on the optimal threshold value. The results of this experiment can be found in figure 2.

Given the optimal threshold value for each logging level we determined the effect of changing the update selectivity for a transaction on the maximum transaction throughput. (See figure 3).

The transaction throughput for tuple logging degrades more quickly as a function of update selectivity than the other methods. This is caused by the logging overhead during normal processing. We expected that for low values of the update selectivity, the recovery mechanism based on tuple logs would beat a recovery mechanism based on XRA-R or XRA-F logging. However, even for update selectivity values, where only a single tuple was updated, XRA-R and XRA-F logs give the highest performance. This effect can be explained by considering that the transaction throughput is dominated by the logging

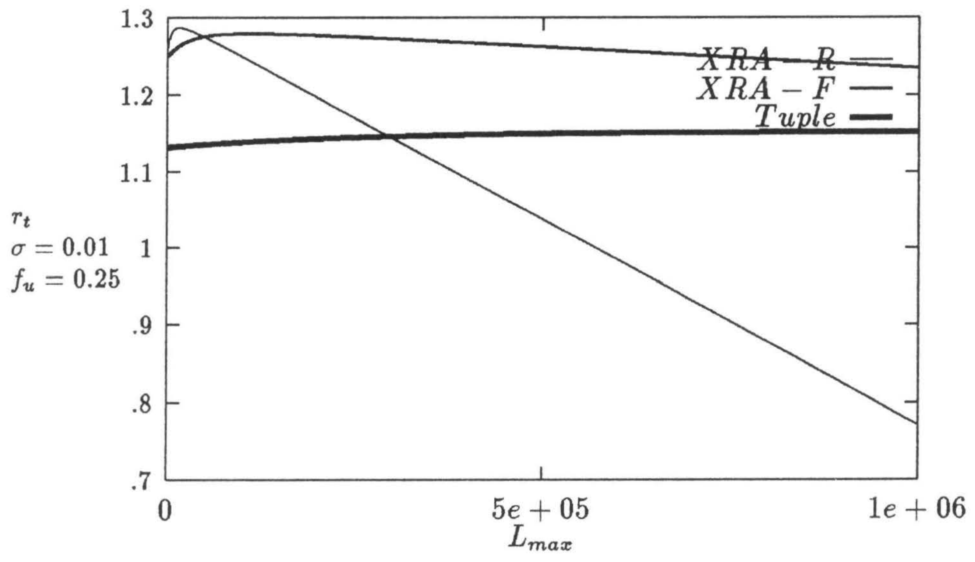


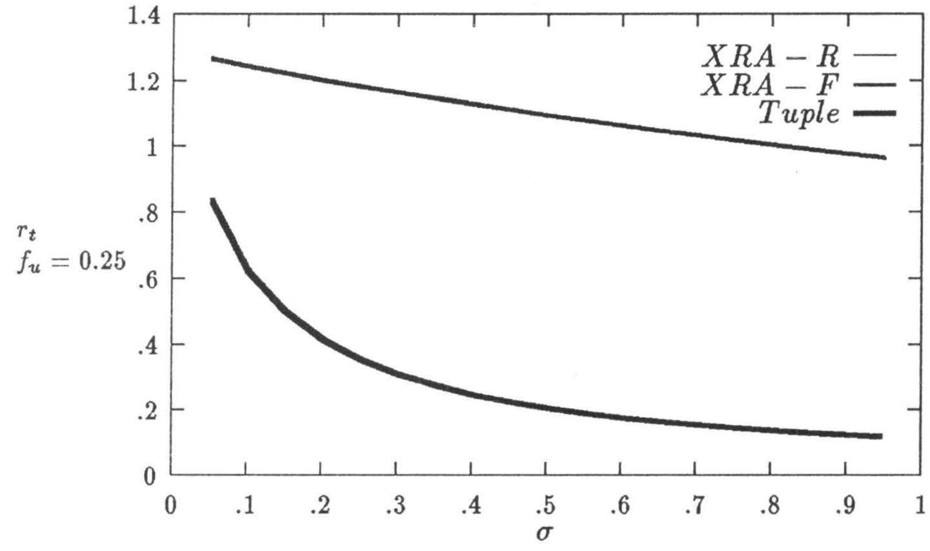
Figure 2: The transaction rate as a function of L_{max}

overhead during normal processing. The replay overhead, which is higher for XRA-R and XRA-F logging, is negligible for large values of MTBF. Experiments with values for the MTBF, which were an order of magnitude smaller, however, showed that tuple logging neither beats XRA-R logging.

By varying the ratio f_u between read and update transactions, we can get an impression of the overhead involved in logging. The results on transaction rate can be found in figure 4. It highlights the considerable influence of the read/update transaction ratio. To find out whether this is should be attributed to the logging overhead or only by the amount of tuple updates, we have run the same experiment with the overhead for logging and checkpointing set to zero. This situation corresponds with a database system equipped with safe ram and a separate checkpoint processor. The result of this experiment is shown in figure 5. As expected, tuple level logging performs under these circumstances (only marginally) better than XRA-R and XRA-F level logging. The maximum transaction throughput, however, is hardly influenced. This indicates that the overhead is primarily determined by the tuple updates and the overhead caused by transactions failures. Although this effect is at first sight in conflict with the results reported in [Eic87], the different findings can be explained. In that paper only tuple level logging was considered. If we only look at the effect of stable memory on the transaction throughput for tuple level logging, we observe a similar increase of transaction throughput.

6 Conclusion

In this paper we have considered the impact of choosing a higher logging for memory resident database systems. The cost model, although very simple, indicates that for trans-



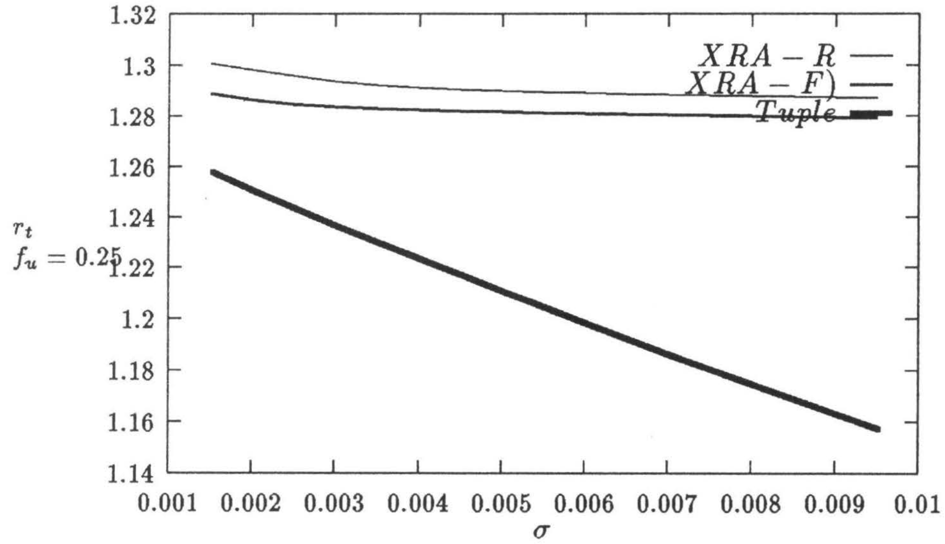


Figure 3: The maximum transaction rate as a function of the update selectivity σ.

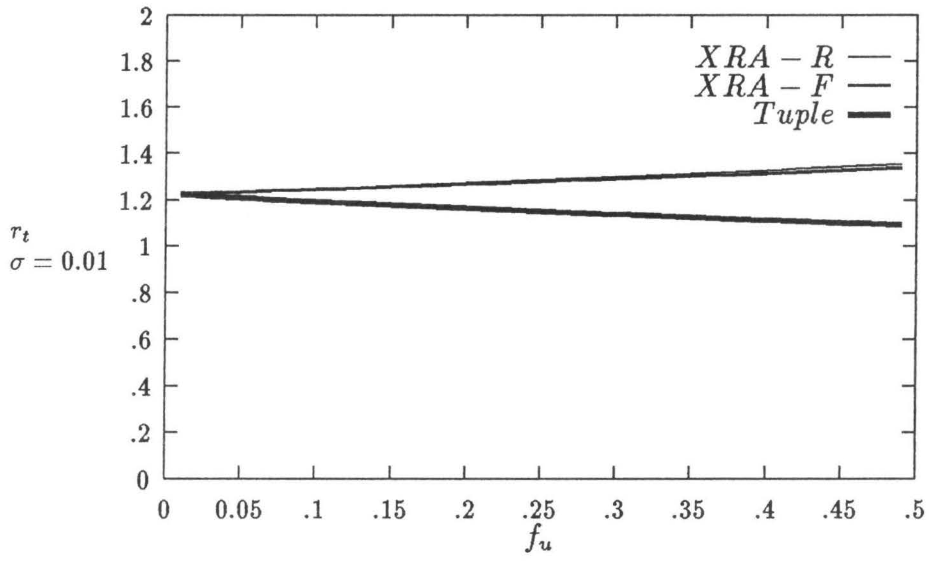


Figure 4: The maximum transaction rate as a function of the read/update transaction ratio, f_u.

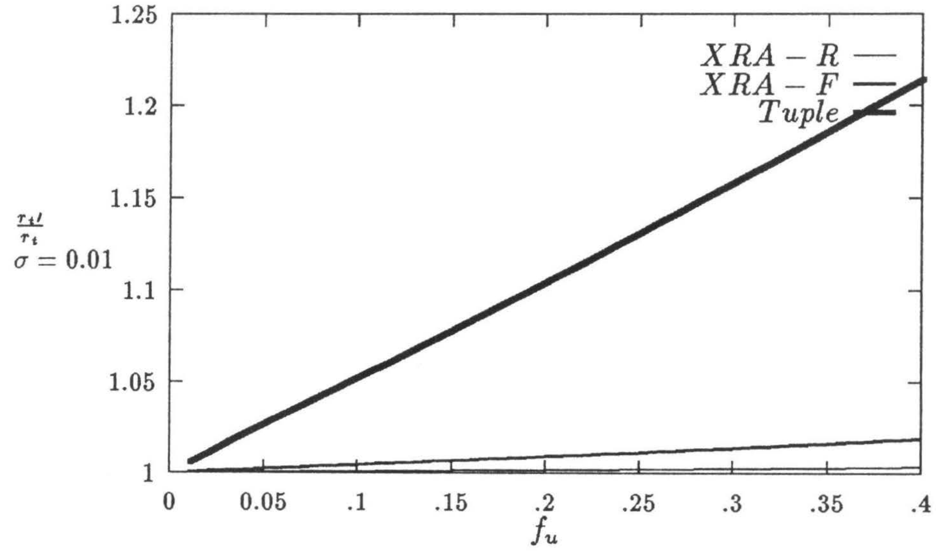


Figure 5: The ratio $\frac{r_{t'}}{r_t}$, where $r_{t'}$ represents the maximum transaction rate in a stable memory environment

actions with a high update selectivity, logging at an algebraic level does improve the transaction throughput over tuple level logging. This method results in a performance gain of about the same order of magnitude as using stable memory to keep the log tables, and parallel checkpointing hardware. The reason is that the cost for log replay after a system crash has decreased considerably, when the database is stored in memory.

In the current implementation of PRISMA, the recovery mechanism is based on logging at the tuple level. Provided that the evaluation of the cost model using the PRISMA architecture does not lead to contradictory results, the next version of PRISMA should seriously consider XRA logging.

7 Acknowledgements

We would like to thank all persons involved in the PRISMA project for the stimulating cooperation and the effort they have put into making this project a success.

References

[AD85] Rakesh Agrawal and David J. DeWitt. Recovery architectures for multiprocessor database machines. In *Proc. SIGMOD*, 1985.

[Ame88] P. America. Language definition of pool-x. PRISMA document Doc. Nr. 350, Philips Research Laboratories, September 1988.

[CBDU75] K.M. Chandy, J.C Browne, C. Dissly, and W.R. Uhrig. Analytic models for rollback and recovery strategies in database systems. *IEEE Transactions on Software Engineering*, 1, March 1975.

[CKKS89] G. Copeland, T. Keller, R. Krishnamurthy, and M. Smith. The case for safe ram. In *Proc. of the 15th International Conference on Very Large Databases*, 1989.

[CP84] S. Ceri and G. Pelagatti. *Distributed Databases, Principles and Systems.* McGraw-Hill, 1984.

[DKO$^+$84] D.J. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebreaker, and D.Wood. Implementation techniques for main memory database systems. In *Proc. ACM SIGMOD Conference*, pages 1–8, June 1984.

[Eic87] Margaret H. Eich. A classification and comparison of main memory database recovery techniques. In *Proc. of the 1987 Database Enginering Conference*, pages 332–339, 1987.

[HR83] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *Computing Surveys*, 15(4), December 1983.

[KAH+87] Martin L. Kersten, Peter M.G. Apers, Maurice A.W. Houtsma, Erik J.A. van Kuyk, and Rob L.W. van de Weg. A distributed, main-memory database machine. In *Proc. of the Fith International Workshop on Database Machines*, pages 353–369, October 1987.

[LC87] Tobin J. Lehman and Michael J. Carey. A recovery algorithm for a high-performance memory-resident database system. In *Proc. SIGMOD*, 1987.

[Mul87] H. Muller. Hardware aspects of fault tolerance. PRISMA document P121, University of Amsterdam, June 1987.

[Reu84] Andreas Reuter. Performance analysis of recovery techniques. *ACM Transactions on Database Systems*, 9(4):526–559, December 1984.

[WG89] A. Wilschut and P. Grefen. Xra definition. PRISMA document P465, Twente University, September 1989.