Engineering a Main-Memory DBMS

Carel A. van den Berg
Martin L. Kersten
CWI
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

The declining cost of memory chips in the early '80s has sparked off several research projects on using main memory as the primary storage medium for database management systems. One of these projects was the PRISMA project. This paper reports on three studies performed at CWI on design issues of main memory database management systems: index structures, relational operations and database recovery techniques.

1 Introduction

The PRISMA project [1] has been a large-scale research effort in the design and implementation of a highly parallel machine for data and knowledge processing [1, 2, 3]. It ran from October 1986 until October 1990 and it involved about thirty persons distributed over several research groups in the Netherlands. The database research group of CWI worked closely with the database group at the University of Twente on the realization of the database processing component, i.e. a distributed main-memory database management system (MMDBMS), called PRISMA/DB.

The starting point for our undertaking were the outlooks for large distributed processing platforms to handle complex relational database applications. We therefore set out to design a database machine, i.e. a dedicated (hardware) software solution to solve the performance and storage problems encountered in this field. The boundary condition imposed upon the project was the programming language POOL [4] as the vehicle for software construction.

The database research challenges posed by the POOL programming environment were roughly separated into 'how can we exploit large main-memories for database processing' and 'how to distribute data and synchronize distributed query processing'. The latter is researched at the University of Twente and the former at CWI. Together we experienced the impact of an evolving parallel object-oriented programming language on a novel hardware platform for the development of a 45,000 lines program.

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

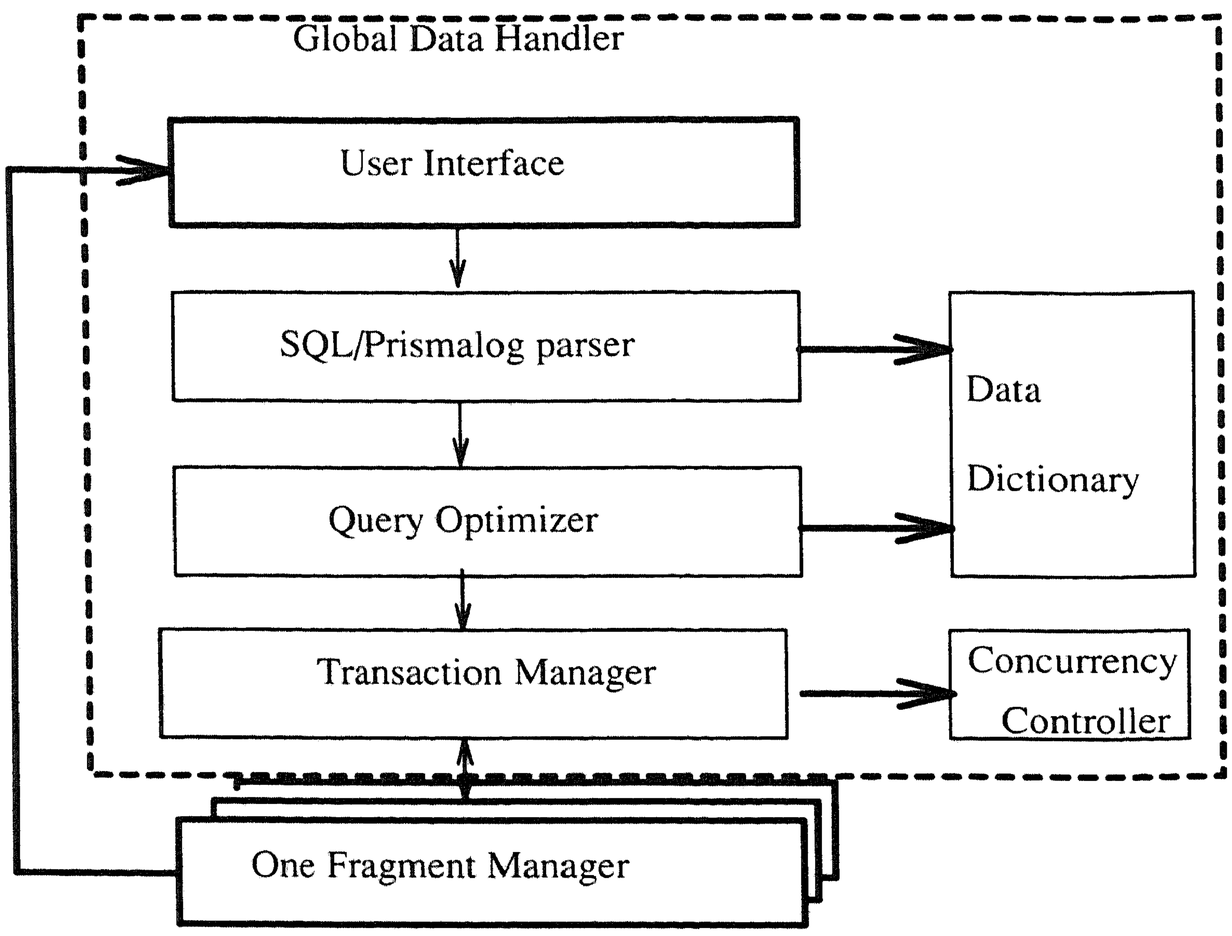

FIGURE 1. The PRISMA Database Architecture

The remainder of this paper is a short account of the research activities within PRISMA/DB undertaken at CWI. More details on the remaining database aspects, programming language and PRISMA machine can be found in [5, 6, 7], respectively. We start with a short introduction to the overall system architecture and its design rationale in Section 2. Following, Section 3 describes engineering results that leads to an efficient use of main-memory for database storage and that improves response time through preprocessing database scan operations. Section 4 deals with the stability issues of a main-memory DBMS, such as database logging and recovery. We conclude with a short summary and an outlook on future research activities.

2 Overview of PRISMA/DB

In this section we introduce the architecture of the PRISMA database system, called PRISMA/DB. An overview of this architecture is given in Figure 1. It shows the functional components, such as a User Interface, Query Parser, Query Optimizer, Transaction Manager, Data Dictionary, Concurrency Controller, and One-Fragment Manager. These components are implemented as communicating processes, i.e. a large collection of POOL objects. The arrows in the diagram represent the major message streams. A brief indication of their role is given in Section 2.1 below.

Based on this general architecture, we indicate the major design issues that had to be tackled in Section 2.2. A more detailed rationale for the PRISMA/DB architecture can be found in [2].

2.1 The PRISMA/DB architecture

The PRISMA/DB database system is designed as a distributed relational database system. In particular, the database relations are considered fragmented and stored in separate data managers, called One-Fragment Managers (OFM), using a horizontal fragmentation rule. Such a fragmented database permits a distributed query processing scheme that can exploit parallelism by running (sub-)queries against the fragments in parallel. A proper fragmentation rule leads to both linear speed-up for many database queries, as a better transaction load distribution.

The user input comes in two forms: SQL or PRISMALog statements. SQL is a standardized relational database query language commonly available in the marketplace. PRISMAlog is a logic-based database query language, in many respects similar to Datalog. Statements written in these languages are translated by a parser into an intermediate language, called eXtended Relational Algebra [8]. Apart from the ordinary relational algebra operators, such as select, join, group, and project, XRA offers operators for dealing with recursive queries and operators to control parallel execution of sub-queries.

The Query Optimizer transforms these XRA statements to a semantically equivalent, but less costly XRA statements using symbolic optimization and heuristic cost functions. Moreover, the optimizer uses the fragmentation rules to re-phrase the query in terms of operations on the relation fragments. In the sequel we refer to these two forms of an XRA query as the XRA-R (on relations) and the XRA-F (on fragments) expression. Thus, the input to the query optimizer is an XRA-R statement to be transformed into an XRA-F statement.

The Transaction Manager takes the XRA-F transaction, requests database locks for the fragments involved and it passes the XRA-F expressions to the individual OFMs. Before a transaction commits, the Transaction Manager checks the integrity constraints to ensure database integrity using a differential method [9, 10]. Furthermore, the transaction manager coordinates logging and system recovery.

The individual OFMs execute the XRA-F statements on their fragment data. For example, each XRA-F update operation, which is set oriented, can result in a sequence of tuple insertions and deletions.

In addition there are two components to administer the system data, called the *Data Dictionary*, and to regulate concurrent access, called the *Concurrency Controller*. The former contains the description of the relational schema, domain information, constraints, sizes, and locations of relation fragments, etc. The latter implements a two-phase locking policy to regulate multi-user access.

2.2 OFM design issues

The OFM developed in our group differs from traditional data managers in one important aspect, namely, we assume sufficient main memory to hold a relation fragment. This requires a complete new design, because in disk resident database systems the performance is largely determined by the number of disk accesses for query processing. Instead, in a parallel main-memory database management system, the query performance is determined by the number of CPU cycles and the communication cost for transferring intermediate results

from one data manager to another. In this paper we focus on main-memory database structures, scan operations, and recovery management.

A common database operation is searching for tuples having a particular attribute value. To speed up this operation, an index is maintained for this attribute. Commonly used data structures in disk resident database systems are linear hashing, and B-trees. These data structures are specially designed for reducing the number of disk accesses. Because this property is less important for memory resident databases, the data structures used for indices were re-evaluated.

As index maintenance is not without cost, there will not always be an index available for speeding up a search operation. Moreover, the search predicate can be an arbitrarily complex expression over the tuple attributes, which makes them less useful. In these situations a scanning operation is used to evaluate the predicate for the fragment tuples. Usually, the scan operation interprets the predicate for each tuple. As this predicate does not change during the evaluation, there is some room for optimization. Two methods have been considered for reducing this interpretation overhead by vectorizing the scan operation and by compiling the tuple predicate. An orthogonal optimization, an adaptive method, is based on sampling the relation to be scanned first, so as to determine the selectivity factors. Thereafter, the expression is transformed according to minimize work.

In database management systems stability of the data is an important issue. Because main memory is volatile in nature, disks have to be used to obtain data stability. However, as the disk is only used as a backup store for the database, it suffices to access the disk only for update transactions and during database recovery. Over the last few years several algorithms have been proposed for logging and recovery of main memory database systems. Not all were applicable to the PRISMA machine architecture. Therefore, we developed our own approach to logging and recovery based on two concepts: log reduction and parallel log replay.

3 Engineering an OFM

In this section we describe efficient search data structures and processing techniques for database scan operations. Both topics illustrate how the constant (CPU+memory) cost associated with well-known algorithms can be improved significantly with fairly simple techniques.

3.1 Data structures

Using main memory as the primary storage for the relation fragment puts some constraints on the data structures for tuple indexes. In particular, the data structure should have a low storage, search, and update cost. Early experiments in this area were conducted by Lehman [11], who studied several main-memory data structures based on hashing and search trees. He compared their storage cost, search cost, and update cost for a representative query mix. Of the hash-based methods (modified) linear hashing [12] turned out to have the lowest update cost and the highest storage efficiency. Of the search tree based methods

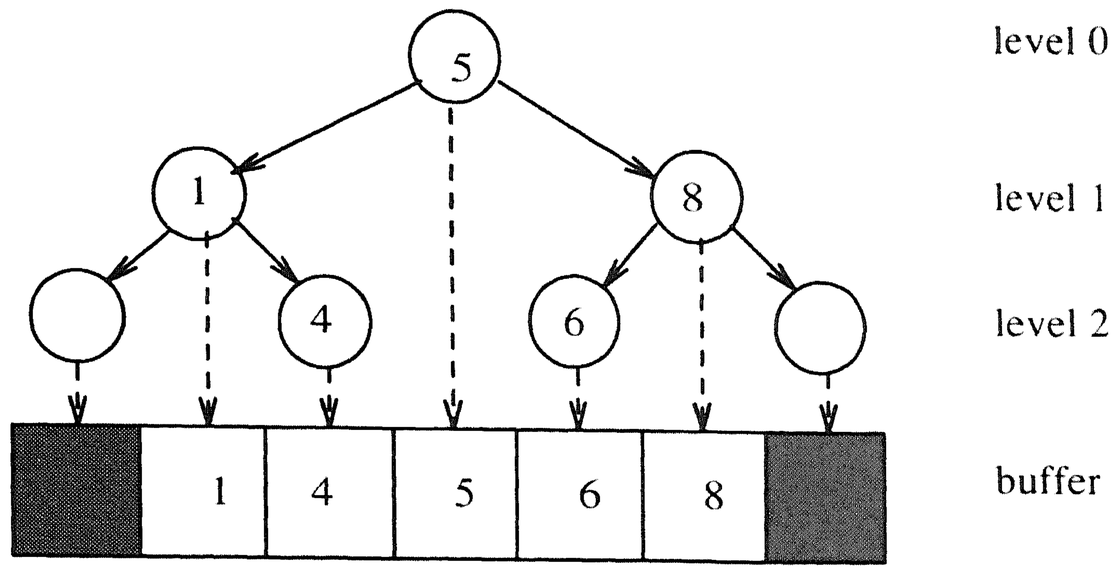

FIGURE 2. The V-tree data structure

a data structure called the T-tree (a variation of an AVL tree) was the best choice.

A recent paper [13] introduces an alternative implementation for search trees, called the V-tree, and compared its performance with a heap, sorted heap, binary tree, and AVL-tree data structures. The V-tree is a compact representation of a (partially) balanced binary tree, using a fixed-size storage array and implicit pointers. The top of a k-level tree is mapped on location $2^{k-1} - 1$ of the array. The children of a node at level l are located at an offset of $\pm(2^{k-l} - 1)$. (See Figure 2.) This mapping ensures that for low load factors, insertion of a new element requires only a local rebalancing of the tree. Rebalancing the V-tree merely requires shifting data in the array. Due to this mapping policy, the empty slots are distributed over the array, thus reducing the number of shifts required after each insert. Measurements show that for load factors below 75%, the insert time for the V-tree is lower than that of an ordinary pointer implementation of a binary tree. Furthermore, as the maximum number of levels in the tree is a priori known, it is possible to use loop unrolling to speed up the search and update procedures. This technique greatly improves the performance at the cost of a logarithmic code expansion.

A disadvantage of the V-tree is that it is static; the maximum number of elements that can be stored is fixed up front. A dynamic data structure can be obtained by combining the T-tree and V-tree data structure. In the context of the PRISMA project we considered two dynamic data structures, TV-tree and TVO-tree, based on a combination of the T-tree and V-tree. The primary objective was again to obtain a data structure, which is both storage efficient and fast. We have measured the insert time, search time, and load factor for the T-tree, TV-tree, and TVO-tree. The storage cost is expressed as a load factor, which is defined as the fraction of the data size in bytes and the total amount of bytes required for the data structure. For a binary tree, for instance, the load factor is $\frac{1}{3}$, because each node consists of two pointers and one data item (assuming simple data items).

The T-tree is a balanced binary tree, which holds many elements in each node

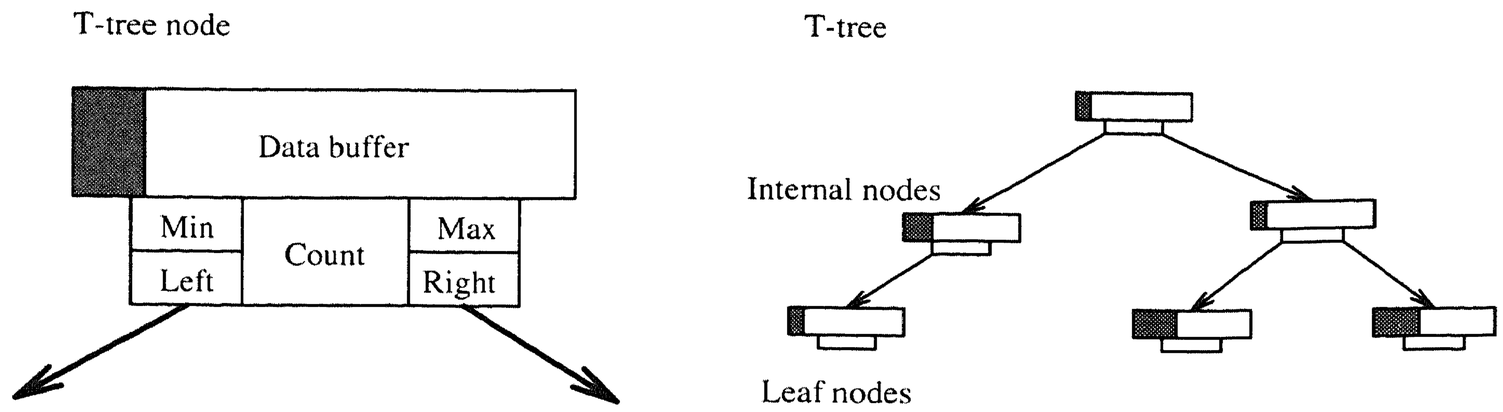

FIGURE 3. The T-tree data structure

of the tree. Each node contains control information, such as the minimum and maximum element stored, and the occupation count. (See Figure 3). All the values contained in the left subtree of a node are smaller than the minimum value and all the values from the right subtree are larger than the maximum value. Whenever the occupation count exceeds a threshold, the node is split and the elements are distributed over the two nodes, possibly followed by a tree rebalancing operation. The elements in the data buffer of a T-node are kept sorted to speed up update and search operations.

The TV-tree uses V-trees as the data structure for the internal nodes. Since the load of a T tree can be rather low, the TVO tree is introduced that uses an overflow area of half the size of the V-tree. That is, if a node reaches the load threshold, an overflow V-tree is allocated. Only if the load of this overflow node reaches the threshold, a node split operation is performed. This approach guarantees that the load of the leaf nodes is at least 50%. If during a node search, the element is not found in the primary V-tree, the overflow V-tree is searched. This implementation is therefore less efficient than the TV-tree.

A few experiments have been undertaken with an implementation of the T-tree, TV-tree, and TVO-tree. Interesting measures are the insert time, search time, and load factor. Figures 4 and 5 present these measures as a function of the number of elements. The load threshold in these experiments has been set to 90%. The TV-tree reaches the same storage efficiency as the T-tree, but gives a better search and insert performance. It turns out that the TVO-tree implementation is also slower than the T-tree, despite the use of V-trees as internal data structures. However, it provides the best storage efficiency (72%).

The load threshold determines the load of the data structure, which is used in the T-tree nodes. A high average load results in a high insert cost. For the V-tree we have seen that a load average below 75% still gives acceptable update performance. In Figure 5 the average load as a function of the load threshold is presented. For the three data structures the load average remains below 75%. The insert time and search time are practically not influenced by the load threshold.

Finally, Figure 6 illustrates the influence of the buffer size on the search time and insert time. By varying the buffer size the search and update cost can be shifted from the T-tree to the V-tree. Obviously, the insert cost increases with

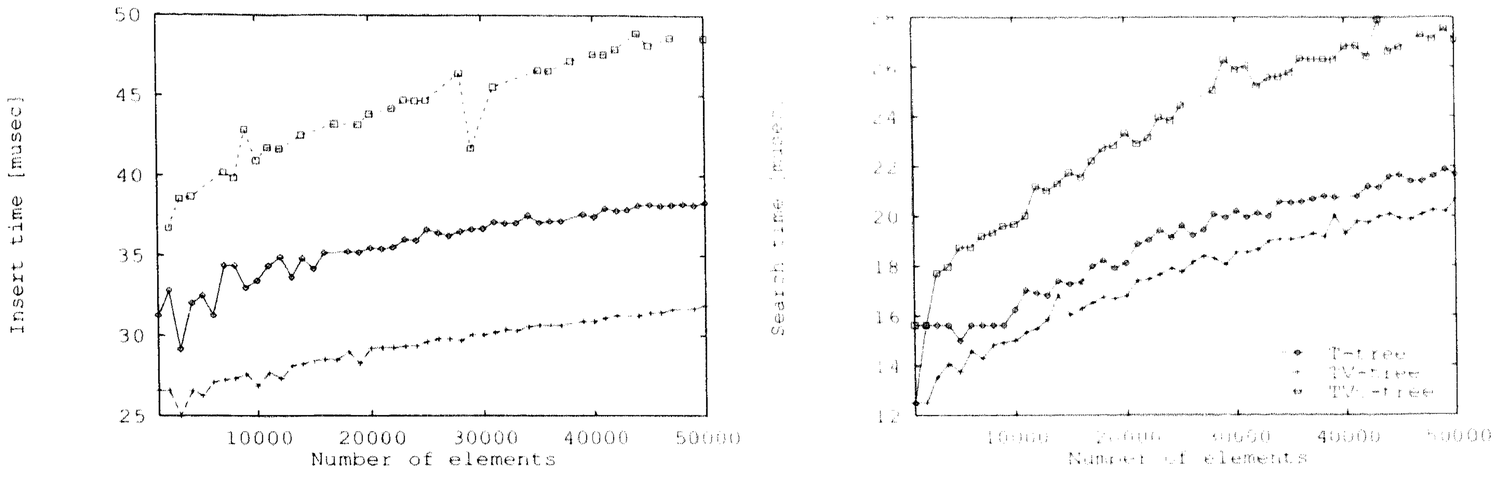

FIGURE 4. Insert time and search time as a function of size

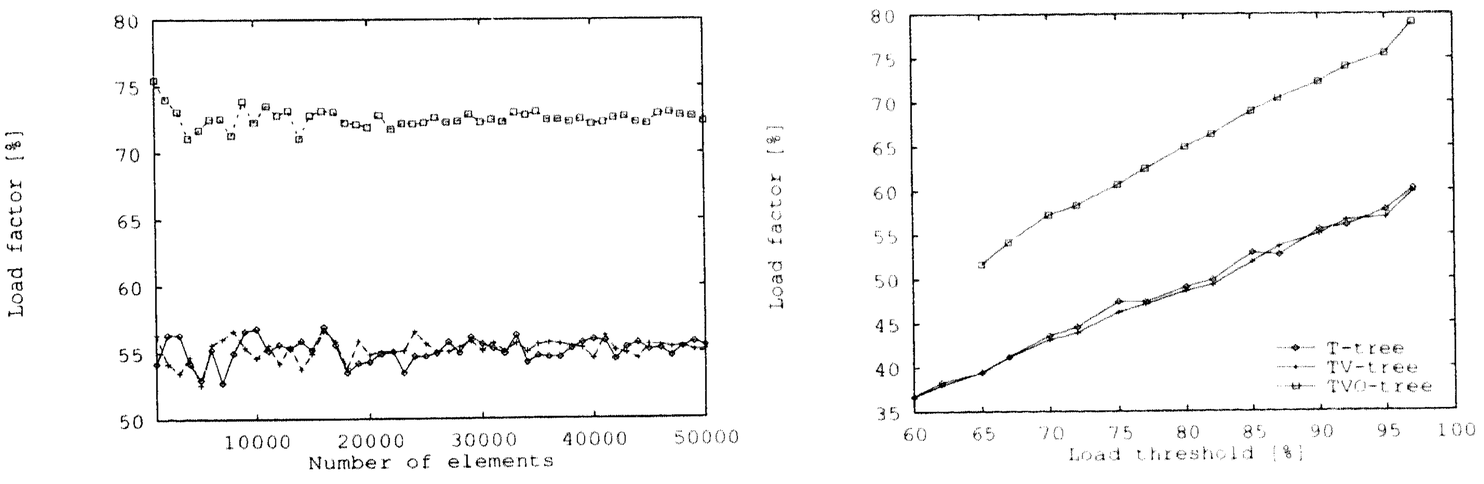

FIGURE 5. The load factor as a function of the size and the load threshold

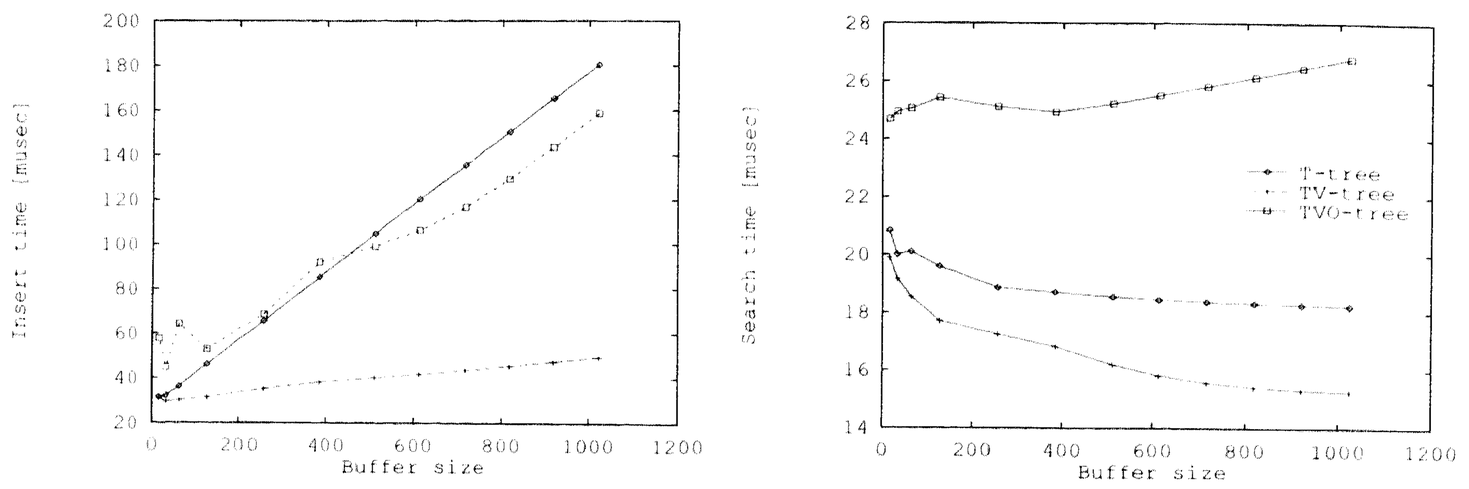

FIGURE 6. Insert time and search time as a function of the buffer size

the buffer size. An insertion in a V-tree or sorted heap results in shifting the data in the buffer. As the buffer grows, the average number of shifts increases also. Insertion of a T-tree node only involves the allocation and initialization of the node, which has a constant cost. However, the update of a T-tree inner node is expensive, due to its sorted-heap data organization.

The search time also decreases with the buffersize for the T-tree and TV-tree data structure, because searching a sorted heap or a single V-tree is cheaper than searching through a T-tree. The search time for the TVO-tree, however, increases slightly with the buffer size. Again, the fine tuning of the search using the V-tree representation reduces the search cost as compared to a traditional T-tree implementation and its overflow V-tree.

3.2 The scan operation

In [14] attention has been given to relational database operations in main memory, but in particular they have concentrated on join algorithms. In [15] the influence of data structures on the performance of all relational operations is investigated. In this section we summarize the results of a performance study on the influence of implementation techniques for relational operations, that aim at reducing the memory traffic. We focus on the scan operation to illustrate, what can be reached by 'clever' programming. The scan operation visits all the tuples and returns those that satisfy the selection predicate.

The common implementation of the scan algorithm is to represent the predicate by an operator tree and recursively interpret this tree for each tuple in the relation. Its performance can be improved with the following techniques:

- By changing the representation of the tree (Compilation). This reduces the memory traffic involved in interpreting the selection condition, but it does not reduce the number of data references.
- By vectorizing the operation, so that the tree is only traversed once (Vectorization). The operators in the selection condition are replaced by vector

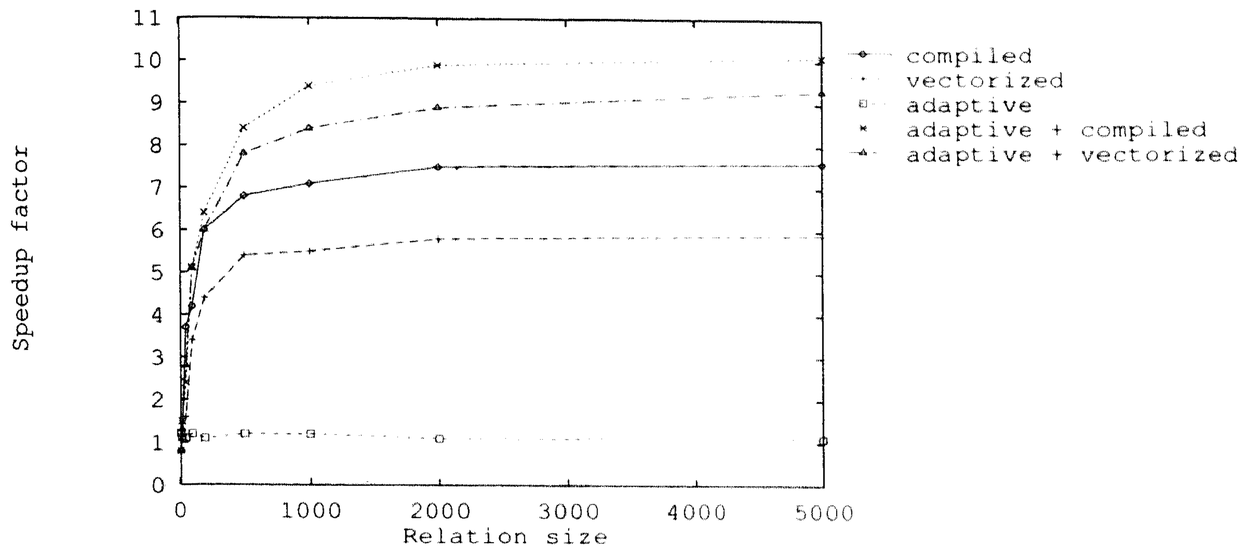

FIGURE 7. Relative execution speed for alternative scan algorithms

and scalar operators. This approach greatly reduces the number of tree references.

- By transforming the tree to reduce the number of data references (Adaptive). Therefore, the tree is weightbalanced on selectivity. To measure the selectivity, the expression is first completely evaluated for a random selection of the available tuples. Then the tree is transformed using this selectivity information and the adapted expression is evaluated for the remaining tuples using vectorization or compilation.

In the experiment the predicates were built out of the operations $\vee$, $\wedge$, $\neg$, $>$, $<$, $=$, $+$, $-$, $\times$, and $/$, and the attribute names and constants. The constants are chosen such that the selectivity of a subexpression can be controlled.

These expressions were evaluated against the Wisconsin benchmark database using the interpretation, compilation, or vectorization method, both with and without the adaptive method.

In a few experiments we measured the performance increase for the compilation and vectorization approach with the basic mechanism (Figure 7). The compilation approach showed a performance increase by a factor 3 - 9, while vectorization showed a gain by a factor 4 - 7.

The vectorization approach and compilation approach reduce the cost for interpreting the selection condition for each tuple. Both methods have a limited amount of start-up overhead. The compilation process requires traversing the selection condition and code generation. The vectorized approach requires memory allocation to store the intermediate vectors. This also requires a single condition traversal to find out the required amount of storage.

The adaptive process turns out to be too costly in most cases. To find out the selectivity of the subexpressions, the selection condition is first completely evaluated for a percentage of the tuples. The number of tuples satisfying the subexpression determine the weights of the subexpressions. Finally, the tree is transformed, so that the subexpressions with the lowest weight are evaluated

first. This method reduces the number of subexpression evaluations and therefore also the number of data references. If 1 % of the tuples is used to determine the hit ratios of the subexpressions, then there is a performance loss over the base algorithm and a slight performance gain over the vectorized algorithm.

The experiments show that with standard optimization techniques like vectorization and compilation, a performance increase of a factor 5 can be obtained. The adaptive method turned out to be only favourable in combination with the vectorization and compilation approach. Our experience with applying these techniques indicates that the performance increase is well worth the added implementation effort for both the compilation and vectorization approach.

4 Logging and recovery

The overhead involved in maintaining a log and checkpoint pair has been identified as a limiting factor for the transaction throughput [16]. 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 paid is an increased CPU cost, due to longer log replay. In general, a trade-off 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 [17], using stable log storage [14, 18], and database partitioning [16].

Carey and Lehman [16] introduce partitions to optimize the log and checkpointing operation. A partition is used to store a variety of information, ranging from indices to tuples. The logfile and checkpoint file for a partition contain 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 [16]. Similarly, logging and recovery is performed in PRISMA at the fragment level.

DeWitt [14] 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 multi-processor database machines. In particular it proves to be an effective technique for improving the transaction throughput.

The paper by Eich [18] 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 the 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 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 ignore the evidently good approaches to equip the system with some safe RAM [19]. The software techniques like partitioning and parallel logging can be used, however.

4.1 Recovery architecture

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 [20], we distinguish between transaction failures, system failures, and media failures.

Transaction failures are likely to happen frequently, about 1% - 10% of all update transactions [21]. 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 [22]. For the other causes of system failure we cannot give a reliable estimation. The machine recovers from a system crash by performing a cold restart, which is followed by reloading the database from stable storage. Thus in the event that one processing node fails, the whole system is rebooted. [2]

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.

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. In PRISMA a two-phase commit protocol is used to obtain transaction atomicity [23].

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 recorded on the logfile are replayed. To reduce the replay time, the database is checkpointed once the log size reaches a threshold value. A low threshold incurs too much checkpointing overhead during normal processing. However, if the treshold is chosen too large, log replay during recovery becomes too time

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

consuming [24]. Clearly, we need a threshold value for which the overhead is minimal.

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 OFMs 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 OFMs 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 relationship 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 information. But the cost associated with recovery could become so high that the effective transaction throughput remains low. So, there is a trade-off 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:

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

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

$$update(R, select(R, dept =\text{”}\ CS\text{”}), salary = salary * 1.10)$$
$$commit/abort$$

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 re-executed in the correct order.

The recovery process requires that all OFMs 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.

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]

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

$update(F_i, select(F_i, dept =''CS''), salary = salary * 1.10) \quad i = 1, \ldots, n_f$
$commit/abort$

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 OFMs and replaying the XRA-F log. [4]

Logging at the tuple level is a form of distributed logging. It is basically the technique described in [23]. Each XRA-F statement results in a list of tuple updates. The exact amount depends on the selectivity of the update σ 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.

$update(tuple_i, f(tuple_i)) \quad i = 1, \ldots, \sigma s_f$
$precommit$
$commit/abort$

The OFMs 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 [14].

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 OFMs 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.

4.2 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 trade-off between logging overhead during normal processing and recovery time. In this section, we want

[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. In general, it is unlikely that this subset is smaller than the complete collection of completed transactions on the transaction log.

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

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 [18] 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. 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 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.

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.

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

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

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 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 L_{max}/n_l update transactions, which produce together $k = \sigma n_f n_t L_{max}/n_l$ 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.

	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

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

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.

4.3 Experiments and results

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 $L_{max}/2n_l$ update transactions have to be replayed after a system crash. This is per log record $1/n_l$.

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.

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.

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

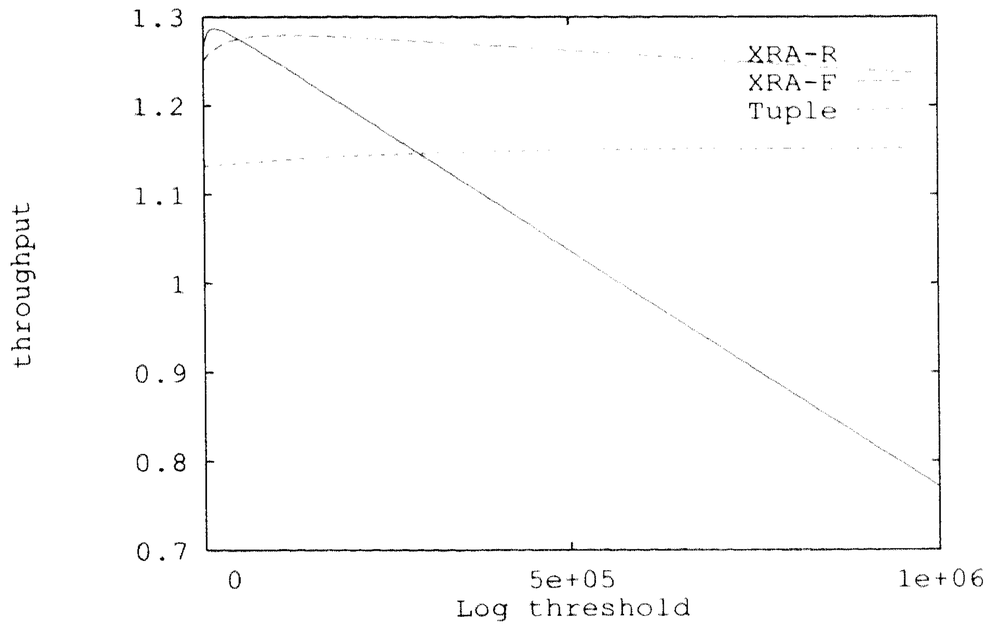

FIGURE 8. The transaction rate as a function of L_{max}

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 derived from the actual PRISMA machine, 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 8.

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 9.)

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 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

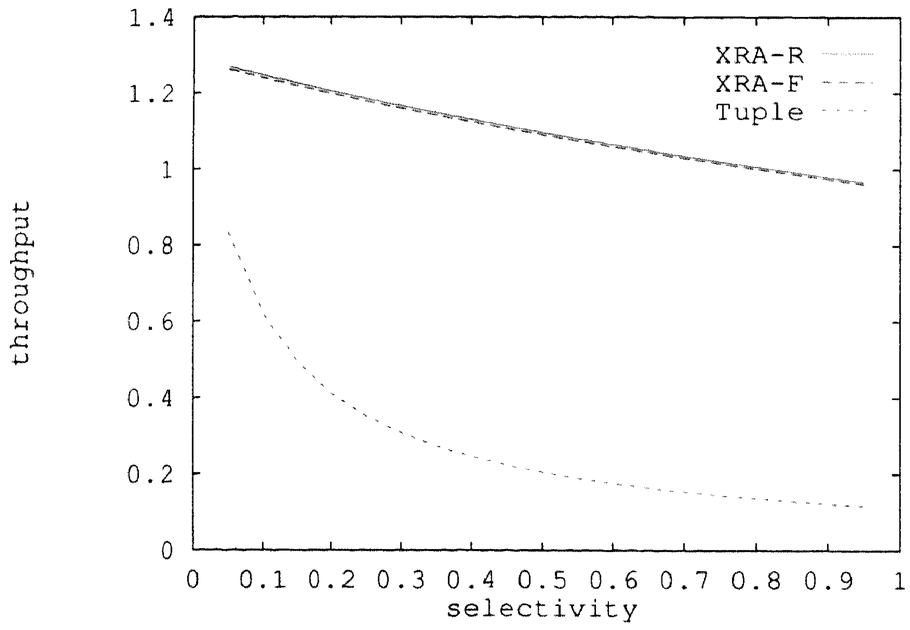

FIGURE 9. The maximum throughput as a function of the update selectivity.

tuple logging still not 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 the transaction rate can be found in Figure 10.

It highlights the considerable influence of the read/update transaction ratio. To find out whether this 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 11. 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 [18], 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.

5 CONCLUSION

In this paper, we have surveyed some of the engineering experiments to obtain efficient solutions to well-known database problems, such as indexing, selection processing and reliability issues. The general conclusion is that with simple techniques significant performance improvements can be obtained over traditional relational DBMS implementation techniques. However, due to separation of concerns between language implementation and database implementation, the results have only been partially incorporated in the prototype POOL implementation of the One-Fragment Manager.

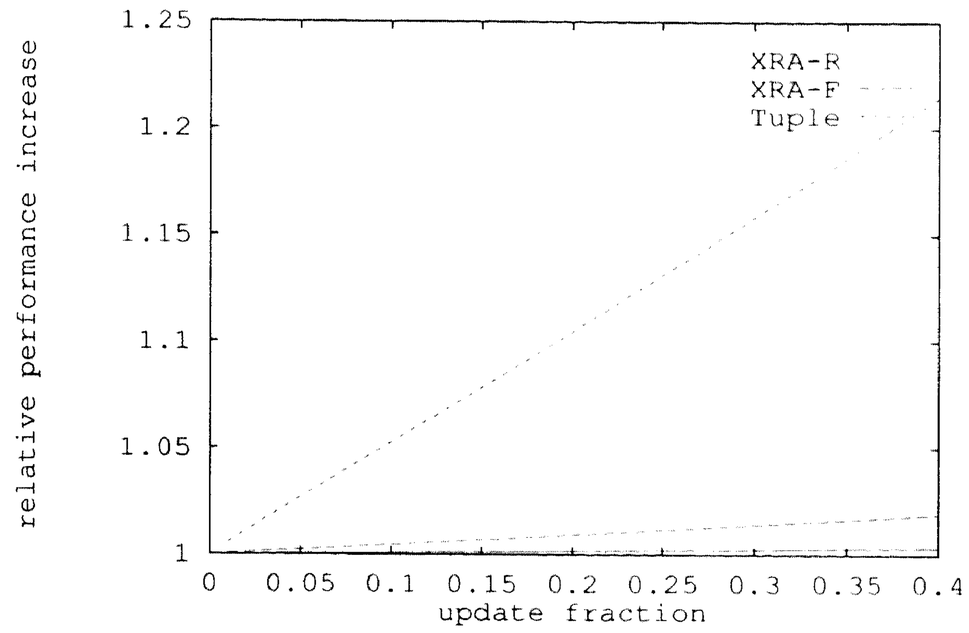

FIGURE 10. The maximum transaction rate as a function of the read/update transaction ratio, f_u.

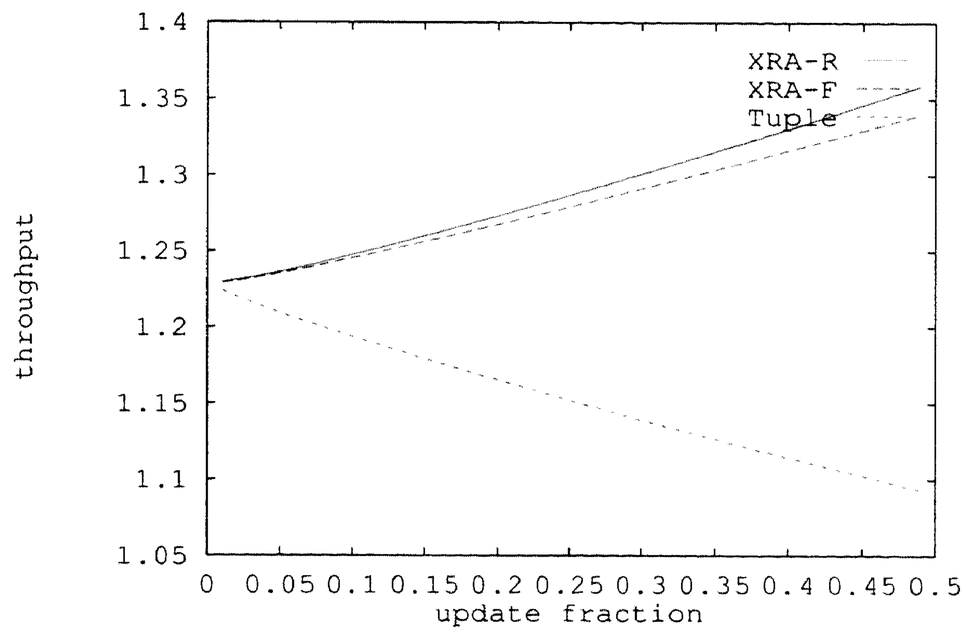

FIGURE 11. The ratio $r_t\prime/r_t$, where $r_t\prime$ represents the maximum transaction rate in a stable memory environment

The prototype OFM uses hash-based indexes, rather than the TV-trees described above. They have been excluded, because it would require extensions to POOL and its implementation. Furthermore, TV-trees are meant to optimize range queries, which are not (yet) recognized by the Query Optimizer.

The expression compilation method has been incorporated in the implementation language and it is widely used in the prototype OFM. In certain cases, where expression compilation was not feasible, such as with specific relational operations, like the group-by operation, vectorization has been used.

The high level logging technique has not been used in the PRISMA prototype, because the primary goal of the PRISMA architecture is to improve query response time instead of increasing transaction throughput.

The PRISMA database machine is now being used at the University of Twente as a platform for further experiments in parallel query processing. The research in the database group at CWI is further focussed on dynamic query processing described elsewhere in this issue.

6 Acknowledgement

The following persons have contributed to the PRISMA project. Carel van den Berg, Marc Bezem, Anton Eliens, Martin Kersten, Louis Kossen, Peter Lucas, Kees van de Meer, Hans Rukkers, Jan Willem Spee, Nanda Verbrugge, and Leonie van de Voort, from Centre for Mathematics and Computer Science. Peter Apers, Herman Balsters, Maurice Houtsma, Jan Flokstra, Paul Grefen, Erik van Kuijk, Rob van der Weg, and Annita Wilschut, from University of Twente. Marcel Beemster, Maarten Carels, Sun Chengzheng, Boudewijn Pijlgroms, Bob Hertzberger, Sjaak Koot, Henk Muller and Arthur Veen, from University of Amsterdam. IJsbrand Jan Aalbersberg, Pierre America, Ewout Brandsma, Bert de Brock, Huib Eggenhuisen, Henk van Essen, Herman ter Horst, Ben Hulshof, Jan Martin Jansen, Wouter Jan Lippmann, Sean Morrison, Hans Oerlemans, Juul van der Spek, Marc Vauclair and Marnix Vlot, from Philips Research Laboratories. Jan Bergstra, Karst Koymans, and Piet Rodenburg, from University of Utrecht. George Leih, from University of Leiden.

References

1. P.M.G. Apers, M.L. Kersten, and H.C.M. Oerlemans. Prisma database machine: A distributed, main-memory approach. In *Proc. Int. Conf. on Extending Database Technology; Venice*, 1988.
2. 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 Fifth International Workshop on Database Machines*, pages 353–369, October 1987.
3. P.M.G. Apers, J.A. Bergstra, H.H. Eggenhuisen, L.O. Hertzberger, M.L. Kersten, P.J.F. Lucas, A.J. Nijman, and G. Rozenberg. A highly parallel machine for data and knowledge-base management: Prisma. Prisma Report P0001, Philips Research Laboratories, Eindhoven, The Netherlands, 1986.
4. P. America. Language definition of POOL-X. Prisma Report P0350, Philips Research Laboratories, Eindhoven, The Netherlands, 1988.

5. A.N. Wilschut, P.W.P.J. Grefen, P.M.G. Apers, and M.L. Kersten. Implementing prisma/db in an oopl. In *Proc. of the 6th International Workshop on Database Machines*, June 1989.
6. E. Brandsma, Sun Chengzheng, B.J.A. Hulshoff, L.O. Hertzberger, and A.C.M. Oerlemans. Overview of the prisma operating system. In *Proceedings of the International Conference on New Generation Computer Systems*, 1989.
7. Marnix Vlot. The pooma architecture. In *Proceedings of the PRISMA Workshop,LCNS 503*, pages 365–396. Springer-Verlag, September 1990.
8. A. Wilschut. Xra syntax. PRISMA Report P280, Twente University, July 1988.
9. P.W.P.J. Grefen and P.M.G. Apers. Integrity constraint handling in a parallel database system. In *Proceedings of the International Symposium on Databases in Parallel and Distributed Systems*, 1990.
10. P.W.P.J. Grefen and P.M.G. Apers. Integrity constraint enforcement through transaction modification. In *Proceedings of the International Conference on Database and Expert Systems Applications*, 1991.
11. Tobin J. Lehman and Michael J. Carey. Query processing in main memory database management systems. In *Proc. ACM SIGMOD Conference*, pages 239–250, May 1986.
12. W. Litwin. Lineair hashing: a new tool for file and table addressing. In *Proceedings of the 6th International Conference on Very Large Databases*, pages 212–223, 1980.
13. M.L. Kersten. Using logarithmic code-expansion to speedup index access. In *Foundations of Data Organization and Algorithms*, pages 228–232. INRIA, Springer-Verlag, June 1989.
14. 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.
15. Tobin J. Lehman and Michael J. Carey. A study of index structures for main memory database systems. In *Proc. of the Twelth International Conference on Very Large Data Bases*, pages 294–303, August 1986.
16. Tobin J. Lehman and Michael J. Carey. A recovery algorithm for a high-performance memory-resident database system. In *Proc. SIGMOD*, 1987.
17. Rakesh Agrawal and David J. DeWitt. Recovery architectures for multiprocessor database machines. In *Proc. SIGMOD*, 1985.
18. Margaret H. Eich. A classification and comparison of main memory database recovery techniques. In *Proc. of the 1987 Database Engineering Conference*, pages 332–339, 1987.
19. 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.
20. Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *Computing Surveys*, 15(4), December 1983.
21. Andreas Reuter. Performance analysis of recovery techniques. *ACM Transactions on Database Systems*, 9(4):526–559, December 1984.

22. H. Muller. Hardware aspects of fault tolerance. PRISMA Report P121, University of Amsterdam, June 1987.
23. S. Ceri and G. Pelagatti. *Distributed Databases, Principles and Systems.* McGraw-Hill, 1984.
24. 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.