PRISMA/DB: A Parallel, Main Memory Relational DBMS

Peter M. G. Apers, *Member, IEEE*, Carel A. van den Berg, Jan Flokstra, Paul W. P. J. Grefen, Martin L. Kersten, and Annita N. Wilschut

***Abstract*—PRISMA/DB is a full-fledged parallel, main memory relational database management system the design of which is characterized by two main ideas. In the first place, high performance is obtained by the use of parallelism for query processing and main memory storage of the entire database. In the second place, a flexible architecture for experimenting with functionality and performance is obtained via a modular implementation of the system in an object-oriented programming language. This paper describes the design and implementation of PRISMA/DB in detail. Also, a performance evaluation of the system shows that the system is comparable to other state-of-the-art database machines. The prototype implementation of the system is ready, and runs on a 100-node parallel multiprocessor. The achieved flexibility of the system makes it a valuable platform for research in various directions.**

***Index Terms*—Parallel, main memory, relational database management system, design and implementation, architecture, query execution, experimentation, integrity constraints.**

I. Introduction

PRISMA/DB is a parallel, main memory DBMS that was designed and implemented during the last five years in The Netherlands by several scientific and commercial research institutions.[1] In the fall of 1986, the PRISMA project was started. The goal of the entire PRISMA project [4] (of which PRISMA/DB is a subproject) is the design and realization of parallel hardware and software to implement the parallel object-oriented programming language POOL, and the implementation of a nontrivial application in POOL. A DBMS was chosen as application. Therefore, PRISMA/DB was designed to be implemented in POOL and to run on the 100-node parallel machine on which POOL is implemented.

In the DBMS group of the PRISMA project, we wanted to study how we could exploit the available resources: 1.6 GBytes of main memory, 100 processing nodes, and a high level parallel programming language. Therefore, the goal of PRISMA/DB is:

the design of a parallel, main memory DBMS that has a *flexible architecture* and that is *flexible* in its *query execution*, so that experiments with the functionality and the performance of the system are possible.

Both for the functionality, and for the performance, there were minimum requirements, such that the resulting prototype can be used for research.

Functionality: The goal is implementing a relational database with the traditional SQL interface and a logical query language, called PRISMAlog, a language similar to Datalog. Furthermore, the database management system should provide concurrency control and support recovery from system failures. The architecture of this system was designed in a modular way to provide opportunities to experiment with the functionality of the system. This facility is currently used for the research in the area of integrity constraint enforcement and query optimization.

Performance: Here, the goal is understanding the influence of parallelism and main memory on performance. The expectation is to obtain a performance comparable to currently available prototype database machines. This performance has to be obtained by both parallelism (100 nodes) and main memory (16 Mbytes per processor). To study the influence of parallelism and the impact of the main memory character of the system, a flexible query execution layer is implemented in the system. Also, special algorithms that exploit the main memory character for the relational algorithms have been implemented. This facility is currently used for the research in the area of parallel query execution.

Obviously, experimentation is a central issue in the project. In many cases, proper design decisions could not be made because of insufficient insight and lack of experience. In that case, the system was set up in such a way that various solutions could be tried out in the final system. This is achieved by a modular architecture and a flexible allocation mechanism of modules to processors [49].

At the starting point of the project in 1986, only few papers on parallel, main memory based database systems on general purpose hardware were available. The low costs of a large main memory system for the end of the 1980's were predicted correctly in [33]. Main memory in 1992 costs about $100K per gigabyte. The potential benefits and problems of an MMDBMS were given in [17] and a single prototype implementation of a shared-store MMDBMS was developed [32]. During the

Manuscript received March 5, 1992; revised August 28, 1992.

P. M. G. Apers, J. Flokstra, P. W. P. J. Grefen, and A. N. Wilschut are with the University of Twente, 7500 AE Enschede, The Netherlands.

C. A van den Berg and M. L. Kersten are with the Center for Mathematics and Computer Science, 1009 AB Amsterdam, The Netherlands.

IEEE Log Number 9204085.

[1]The main partners of the project were: Philips Research Laboratories in Eindhoven, The Netherlands, University of Twente, Center for Mathematics and Computer Science, and the University of Amsterdam.

project's life cycle, an increasing number of papers appeared that address technical issues for MMDBMS implementations. This special issue is its proof of evidence. The development of PRISMA/DB and related studies were influenced by the work on recovery issues [18], [31], parallelism in large-scale comparable (disk-based) systems, such as GAMMA [15], Bubba [10], and HC16-186 [12]. The role of main memory to hold the entire database is getting more support, as illustrated by the shared-store systems XPRS [40], DB3S [8], and the distributed store system EDS [36], [42].

The goals of the PRISMA project were ambitious. Hardware, system software, and the database management system were all developed from scratch. For a period of 4 years roughly 25 people worked on the project; not all of them were directly involved with the database machine. Halfway through the project efficiency problems were discovered with the implementation of the language POOL. After about three and a half years, the first prototype was running on the 100-node multiprocessor system. Since then, pieces of the system are being rewritten to obtain a better performance. Currently a 100K by 10K join of the Wisconsin benchmark runs in 2 s.

Research is now focused on a few topics to investigate the performance and the flexibility of the architecture. We have chosen to extensively investigate the influence of main memory and parallelism on query execution and constraint enforcement. For other components of the system, like the concurrency controller, off-the-shelf solutions were chosen, or, in case of recovery, a more concise study of main memory alternatives led to the implementation of a parallel algorithm. The main research topics are: performance evaluation, parallel join evaluation in a main memory environment, and parallel constraint enforcement. This research has revealed that the main memory character of the system has significant impact on its parallel behavior, that specialized main memory algorithms are possible and profitable from performance viewpoint, and that the fast read-only access to a main memory system allows extensive integrity constraint checks with limited performance penalty. Each research topic is discussed in more detail in this paper.

This paper is organized as follows. The next section briefly introduces the 100-node parallel multiprocessor that is used, and the implementation language POOL-X. Section III first gives an overview of the DBMS architecture and then highlights the following aspects of this architecture: internal representation of queries, parallelism and data fragmentation, transaction management, query execution, and storage and recovery. After that, Section IV illustrates the dynamic aspects of the architecture via the description of an example query execution. Section V describes the performance of PRISMA/DB, and it discusses the relationship between the influence of parallelism and the main memory aspects of the system. Section VI briefly describes the current research in the context of PRISMA/DB, and Section VII summarizes and concludes the paper.

II. Hardware and Software Support

PRISMA/DB is implemented on a parallel multiprocessor, called the POOMA machine. On this machine, a parallel, object-oriented language, POOL-X, and an operating system that supports POOL-X are implemented. This section summarizes the hardware and the essential features of POOL-X.

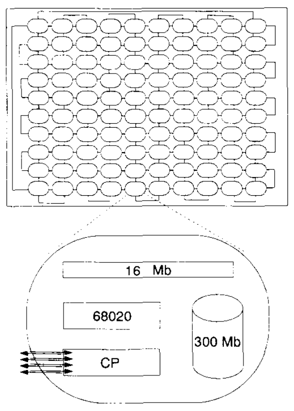


Fig. 1. Hardware configuration of the POOMA machine.

A. The POOMA Machine

The POOMA machine is a shared-nothing, parallel multiprocessor, which consists of 100 nodes. Reference [13] describes its design and the rational behind it in detail. Fig. 1 shows the hardware configuration. Each node consists of a 68 020 data processor with 16 Mbytes of memory, a disk, and a communication processor that links it to 4 other nodes using bidirectional links. The processor memory consists of 4 Mbytes of on-board memory, and a memory extension board containing 12 Mbytes. The use of slower memory extension boards results in the 4 MIPS processors to run at only 1.5 MIPS in practice. Some nodes have an ethernet card that links the system to a Unix host. The nodes are linked together using communication processors that were developed by Philips. Various configurations can be realized. Fig. 1 shows a mesh connection; other configurations, such as a cordal ring connection and a double linked ring connection are also possible. The entire system contains 1.6 GBytes of memory.

B. POOL: A Parallel Object-Oriented Language

The programming language POOL-X [2], [3], [37] is implemented on the POOMA machine, and is used as an implementation language for PRISMA/DB.

As an object-oriented language, POOL-X allows the definition of *objects*, which are functional units of data and methods that operate on the data. In POOL-X, process objects and data objects can be discriminated. Process objects have an individual thread of control, and data objects are used by process objects as data structures. The discrimination between process objects and data objects was made for efficiency reasons.

Parallelism is supplied in a very natural way: conceptually, all process objects that exist in the system execute concurrently. Allocation of two process objects to different processors makes them really run in parallel. Also, objects can be created and deleted *dynamically*. These features turn a POOL-X program in execution into a very flexible structure which allows runtime experimentation with various forms of parallelism.

Objects can *communicate* synchronously and asynchronously. A synchronous message to another object causes the sender to wait for the reply. An asynchronous message does not have a reply. Synchronous communication between objects synchronizes their execution and may, therefore, impede the effective parallelism. Asynchronous communication does not have this drawback. Communication between objects that are allocated to different processors is automatically translated into interprocessor message passing.

POOL-X has some special facilities for the implementation of a DBMS: tuple types can be created dynamically. Also, conditions on tuples can be *compiled* into routines. This feature is used to speed up scan operations in which a condition has to be evaluated for a large number of tuples, like selections and joins.

It should be noted that the language POOL-X was developed and implemented parallel to the design and implementation of PRISMA/DB. This had consequences for the development of PRISMA/DB. About halfway through the project, there were severe performance problems in the POOL-X implementation. As a consequence, we could not evaluate the performance of the first try-out prototype. Also, the POOL-X compiler that is currently used is not yet optimized in detail. This results in the performance of PRISMA/DB not being quite optimal.

III. Architecture

This section presents the software architecture of the PRISMA database management system. First, an overview is given of the global architecture. Next, the most important aspects of this architecture are discussed in detail: the internal relational language, extended relational algebra (XRA), query optimization and parallelism in query execution, transaction management and integrity control, query execution mechanisms, and finally, storage and recovery aspects. Note that this section focuses on the static aspects of the architecture. The dynamic aspects are illustrated in Section IV, where examples of query execution are described in detail.

A. Overview

Fig. 2 presents an overview of the architecture of PRISMA/DB. The architecture consists of a number of components that are implemented as POOL-X process objects. Some components are instantiated several times in the system, others are central: they have one instantiation that serves the entire DBMS. The architecture is dynamic: components can be created and deleted dynamically, according to the use of the system. Each component has a well-defined functionality, and much effort was put in the design of the interfaces between the components. This modularity through function separation and high level interfaces is an important characteristic of the design of the system [49]. As a result, the flexibility in the system architecture allows experiments with functionality.

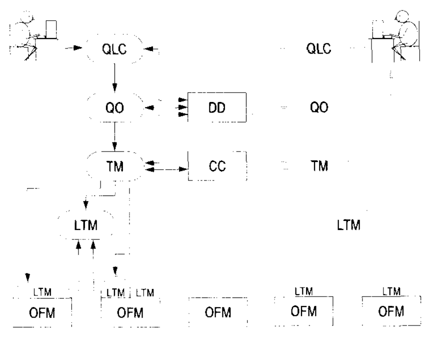


Fig. 2. Global architecture PRISMA/DB.

The rectangles in Fig. 2 represent permanent components, i.e., components that live as long as the system. The ovals represent transient components belonging to one user session; the life cycle of these components is related to user actions. The dotted ovals show transient components belonging to a second, concurrent user session. The function of the components and the interfaces with other components are described in the following.

Two central components of the system are the *data dictionary* (DD) and the *concurrency controller* (CC). These components are instantiated once in the system. The choice for a central CC and DD was made for simplicity reasons. The DD is the central storage of all metadata of the system, like relation and constraint definitions, fragmentation information, and statistics. The contents of the DD are entirely stored in the primary memory; a disk copy is kept to ensure recoverability. Data definition statements result in changes of the DD tables; these changes are immediately written to the disk. The concurrency controller controls concurrent access to the database. It uses a standard two-phase locking protocol with shared and exclusive locks. Furthermore, it is equipped with a deadlock prevention algorithm.

The query preprocessing layer of the system is formed by the *query language compiler* (QLC) and *query optimizer* (QO) components. As shown in the figure, these components are instantiated once for each user session. The QLC provides an interactive interface to the user and translates queries from the user language into the internal relational language of the system (XRA, see Section III-B). This component offers full fragmentation and allocation transparency to the user [14]. Four different QLC's are available: a standard SQL interface, a logical query interface called PRISMAlog, that allows recursive queries [5], an XRA interface that allows queries in the internal language of the system, and a simple data definition interface via which relations can be created, integrity constraints can be defined, and the fragmentation of relations can be changed. Translated queries are sent to the QO, which optimizes them into parallel execution plans (see Section III-C). The QLC's and the QO's contact the DD to obtain the schema information and statistics needed for the translation and optimization of queries.

The *transaction manager* (TM) forms the execution control layer of the system. This component is instantiated once for each transaction. The TM coordinates the execution of a transaction via an interface between the TM and the query execution layer of the system. Furthermore, the TM contacts the CC to ensure serializability of the transaction; the atomicity and recoverability of the transaction are enforced through a two-phase commit protocol between the TM and the execution layer; the correctness of a transaction is guaranteed through the enforcement of integrity constraints, which are retrieved from the DD. Transaction management is described in more detail in Section III-D.

The data storage and query execution layer consists of the *one fragment managers* (OFM's) and the *local transactions managers* (LTM's). OFM's are permanent; they store and manage the main memory copy of one fragment of a relation in the database, and the logging and checkpointing information that is kept on the disk for recovery. LTM's are transient; they are the relational engines in the system. The LTM's use especially designed main memory algorithms for the relational operations. The query execution layer is described in more detail in Section III-E.

The design of PRISMA/DB allows parallelism between components. If, for example, the QLC and the QO of one session are allocated to different processors, they can work concurrently, forming a pipeline. Also, allocation of the components of a second session to a new (set of) processors yields interquery parallelism on the query preprocessing level. Finally, allocation of OFM's and LTM's to different processors leads to parallel query execution in several forms. This issue is described in Section III-C.

The main interface language between the various components of PRISMA/DB is formed by an extension to the relational algebra, called XRA [23]. This language provides flexible, high level communication between the various query processing layers of the system. The language is discussed in detail in the following.

B. XRA

An XRA is used as an internal representation of queries in the system. A full description of its syntax and semantics can be found in [23]; here the main features are described.

XRA contains the standard relational operations (selection, projection, Cartesian product, join union, difference, and intersection), update facilities (insert, delete, and update), and some extensions like a grouping operation, sorting facilities, and a transitive closure to support recursive queries from the PRISMAlog interface.

Also, XRA offers the flexibility to express a wide variety of parallel query execution plans: an operand can consist of multiple tuple streams that are automatically merged to form one operand, and the result of an operation can be distributed over multiple output streams. This distribution of result tuples can be done in two ways: the result can be replicated over output streams, or a hash- or range-based splitter is applied to split the tuples over the output streams. The use of these primitives to formulate parallel query plans is illustrated in Section IV.

Finally, a simple projection that can only throw away some attributes from a tuple (as opposed to the facility to do, e.g., arithmetic operations on attributes) is added to the language. This operation is used as a cheap filter on tuples before they are sent to another processor to reduce the communication costs. Again, its use is illustrated in Section IV.

C. Parallelism and Data Fragmentation

PRISMA/DB supports parallel query execution. The use of parallelism is completely transparent to the user. The query preprocessing layer of PRISMA/DB, which consists of the QLC's and the QO, translates user queries on the relational level into parallel execution plans on the fragmented database, taking the fragmentation scheme of the stored database into account. This section describes the generation of parallel execution plans. To do so, first the terminology used with respect to parallelism and data fragmentation are introduced.

1) Parallelism: In PRISMA/DB, various forms of parallelism can be used to speed up query execution. The standard terminology for parallelism [11], [49] is used. To be complete, the used terminology is summarized here. Multiple users can use the system concurrently, yielding *interquery* parallelism between their queries. Within a query, *intraquery parallelism* can be subdivided into *interoperator*, and *intraoperator* parallelism. Orthogonal to this distinction, *pipelining* can be contrasted to (pure) *horizontal parallelism.* The term parallelism is often used as a synonym of horizontal parallelism. This paper adopts this habit if no confusion is possible. Horizontal intraoperator parallelism is very commonly used. The term *data parallelism* is often used for this form of parallelism; the number of processors used is called the *degree* of parallelism.

2) Data Fragmentation: Relations in PRISMA/DB are horizontally fragmented across a number of processors. Horizontal fragmentation of data enables parallel execution of operations on the data. For example, to execute a selection on a fragmented relation, it suffices to execute a selection on each of the data fragments.

Because PRISMA/DB uses hash-based algorithms for many relational operations, hash-based fragmentation is used. An arbitrary attribute can be used as a fragmentation attribute. To distribute the tuples in a relation over its fragments, a hash function with a large range is applied to the specified attribute, and the resulting value modulo gives the number of fragments used for the relation, which indicates the fragment where the tuple belongs. So, specifying the fragmentation attribute and the number of fragments pins down the fragmentation. Each fragment can be assigned to an arbitrary processor. The number of fragments that is used for one relation is called the *fragmentation degree* of that relation.

This fragmentation strategy offers the possibility to experiment with fragmentations schemes for one relation that differ in the their degree and fragmentation attribute. Range-based fragmentation is currently not supported. The extension can easily be added, however, as XRA has the facility of range-based splitting a relation.

The fragmentation of a relation and the allocation of the fragments can be specified by the user at creation time. Also, a relation can be refragmented runtime, and the fragments can be reallocated to other processors. This allows experimentation with different allocation and fragmentation schemes in one session.

3) Generating Parallel Execution Plans: User queries are transformed into parallel execution plans by the query preprocessing layer. The QLC takes a query in one of the user languages, and after syntactic and semantic checking, it is translated into XRA on the relational level (XRA-R).

The QO transforms this XRA-R query into a parallel execution plan in XRA-F (XRA on the fragment level). To do so, it retrieves fragmentation information from the DD. Because we are still studying the problem of optimizing complex queries for parallel execution (see Section VI-A), only simple optimizations are used: selections and projections are pushed down as far as possible, and many relational operations can be distributed over unions. This means that the QO will transform a join over the union of the fragments that belong to one relation, into a union over the fragment joins, thus generating a parallel execution plan for a join. The fragmentation information is taken into account in this process: if the operands of a join are fragmented on the join attribute into the same number of fragments, the fragments can be joined to each other directly, otherwise, one or both fragments are redistributed before the join. In the same way, many other relational operations are parallelized. Finally, the QO allocates the operations in the parallel schedule to processors, taking the allocation of the base-fragments into account: e.g., a join of two fragments that reside on different processors is allocated to the processor where the larger operand resides, so that the data transmission costs are minimized.

For the implementation of the QO, a rule-based approach was chosen [29], in which the optimization strategies are stored in a rule base that is attached to an optimization engine. This architecture of the QO facilitates changes in the optimization strategy that is used, so that research results in the area of parallel query processing can easily be implemented. The performance of the optimization process itself is currently not a research issue.

D. Transaction Management and Integrity Control

The PRISMA/DB TM is responsible for the management of one single transaction. The TM has two main tasks. First, it is responsible for creation and control of the transaction execution infrastructure consisting of OFM's and tuple transport channels, and it schedules the execution of the individual operations in a transaction. Secondly, it takes care of the transaction properties: atomicity of transaction execution, correctness with respect to defined integrity constraints, serializability with respect to concurrent transactions, and recoverability. The TM has a modular internal architecture the design of which was inspired by the tasks mentioned above; an overview of the architecture is given in Fig. 3.

Transaction commands coming from the query optimizer are first analyzed. One of the main goals of the analysis is to determine the necessary locks for the execution of the commands. This locking information is passed to the local

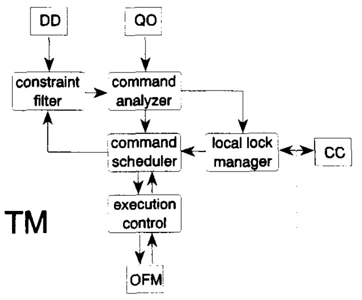


Fig. 3. Architecture of the TM.

lock manager. This module decides whether locks are already owned by the transaction, or have to be requested from the CC. Analyzed commands are scheduled for parallel execution, such that all commands are executed as early as possible. The scheduling takes both the dependencies between various commands and the availability of locks into consideration [22]. Commands that are ready for execution are sent to the execution control module. This module is responsible for the control of the actual execution of commands at the OFM layer of the system. Where necessary, it creates transient LTM's and tuple transport channels to form the execution infrastructure for the commands in the transaction. After having created this infrastructure, it sends the XRA commands to the appropriate LTM's. At transaction commit time, the integrity constraints to be enforced are retrieved from the DD. Based on a syntactic analysis of the update commands in the transaction, only those constraints are retrieved that may be violated by the transaction. The constraints have been translated into XRA commands at definition time by the DD, and can thus simply be appended to the end of the transaction, according to the transaction modification principle[2] [21]. The execution of the constraints can use exactly the same mechanism as normal query execution. In this way, constraint enforcement automatically satisfies the serializability and transaction atomicity requirements. At the very end of transaction execution, a two-phase commit protocol is executed to ensure transaction atomicity, in which the TM acts as coordinator and all OFM's involved in the transaction act as participants.

E. Query Processing

This section describes the query execution layer of PRISMA/DB. This layer consists of the OFM's, which store and manage base data, and the LTM's which are the relational engines of the systems. Fig. 4 shows the organization an OFM-LTM combination in the query processing layer.

An OFM manages one fragment of a relation; it is a permanent component, which is implemented as a POOL-X process object. As such, a fragment of a relation is the unit of data allocation in the DBMS, and the allocation facilities of POOL-X can be used to experiment with different allocation schemes for the fragments of the stored database.

[2]Note the difference with the *query modification* approach [39], where the selection predicates of updates are extended with the negation of constraint predicates.

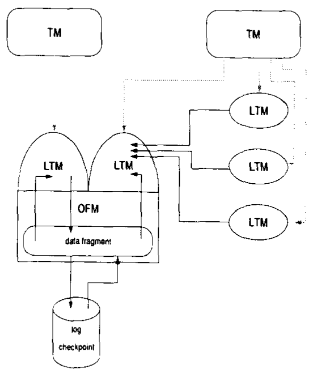


Fig. 4. Local transaction management and its environment.

An LTM is a transient object; it can execute relational operations. Typically, an LTM is created for each fragment operation in a query. Some LTM's are attached to an OFM, in which case they can directly access the fragment that is managed by that OFM; other LTM's are independent and they operate on the results of previous operations in the same transaction. This can be a stream of tuples that is generated by one or more other LTM's, or the stored result of a previous operation of the same LTM. LTM's are also implemented as POOL-X process objects, and they form the unit of parallelism in the query execution layer of the DBMS. Again, the allocation facilities of POOL-X can be used to experiment with various parallel execution strategies for a query.

The query execution layer of PRISMA/DB is designed to allow flexible parallelism: one operand can consist of multiple input streams that are merged by the LTM to form one operand (in Fig. 4, three LTM's produce one operand for the destination LTM). On the other hand, the result of an operation can be distributed over multiple output streams, each with its own destination. One OFM can concurrently be accessed by multiple transactions (only for reading, of course); in that case each TM attaches a private LTM to the OFM.

The main memory character of the system is exploited in the algorithms for relational operations. In general, we can state that a main memory system allows relatively simple algorithms that are not bothered by buffer and cache management problems. Obviously, such a system allows optimizations that only yield performance gain in a main memory environment. For example, [6], [7] describe a study of possible optimizations of operations that scan large numbers of tuples. It was shown that dynamic compilation of expressions that have to be evaluated for a large number of tuples yields considerable performance gain. Therefore, PRISMA/DB heavily uses the dynamic compilation facility of POOL-X.

The architecture of the LTM's allows both pipelining and horizontal parallelism between different LTM's. In PRISMA/DB, we want to study both forms of interoperation parallelism in the context of a main memory system. In [43], [45] it is shown how special main memory algorithms can be used that enhance the effective parallelism from pipelining.

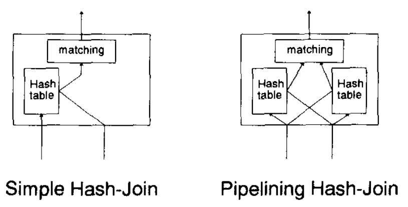


Fig. 5. Simple hashjoin and pipelining hash join algorithm in a main memory system.

These pipelining algorithms aim at producing output as early as possible so that a consumer of the result can start its operation. In particular, [43], [45] proposes a pipelining hash join algorithm. As opposed to the well-known simple hash join algorithm, this symmetric algorithm builds a hash table for both operands (see Fig. 5). The join process consists of only one phase. As a tuple comes in, it is first hashed and used to probe that part of the hash table of the other operand that has already been constructed. If a match is found, a result tuple is formed and sent to the consumer process. Finally, the tuple is inserted in the hash table of its own operand. This algorithm can produce a result tuple as soon as two matching tuples have reached the join LTM. Pipelining algorithms are also possible for many relational operations other than the join operation. Where possible, pipelining algorithms are used for the implementation of relational operations. Reference [48] discusses the influence of using pipelining algorithms on the performance gain from interoperation pipelining.

F. Storage and Recovery

PRISMA/DB is a main memory DBMS; this means that the entire database is stored in the primary memory (RAM) of the system. To make this a realistic assumption, the system must provide a large amount of RAM memory. The POOMA prototype is equipped with a total of 1.6-GB RAM. Furthermore, the scalability of the hardware architecture allows the addition of nodes to increase this amount of memory. The POOMA hardware is not equipped with stable RAM memory, however. As a consequence, the contents of its memory are lost after a system crash. To ensure stability of the database, a stable storage medium is required as backup storage for the main memory database. PRISMA/DB uses the POOMA stable file system for this purpose.

1) Storage: Since PRISMA is designed as a main memory system, the traditional DBMS storage structures for the relations have to be re-evaluated. The OFM is equipped with data structures for handling tuples. These data structures are available to each LTM that is attached to an OFM (see Fig. 4). In particular, tuple layout, index creation, storage preservation, and temporary storage are important for their design. See, for instance, [6], [28], [30] for a comparison of data structures for main memory database systems. The tuple layout is critical for both storage and performance. Tuple lengths for business like applications can be assumed to be less than 4K bytes. Most tuples will be rather short (0.1–0.5 K). Moreover, the use of main memory relaxes the need for physical adjacency of

Person

id	name	age
1	"Paul"	27
3	"Jan"	27
11	"Joost"	6
13	"Carel"	29
2	"Peter"	38
4	"Annita"	36
10	"Martin"	37
6	"Anna"	1

Drinks

pers	wine	#
1	101	2
1	105	5
3	110	2
11	110	1
13	102	7
2	105	5
2	106	3
4	103	4
4	104	6
10	108	2

Wine

id	name	year
100	"Chablis"	1980
102	"Riessling"	1985
104	"Beaujolais"	1992
106	"Bordeaux"	1988
101	"Chablis"	1983
103	"Almaden"	1991
105	"Riessling"	1990
107	"Bordeaux"	1980
108	"Bourgogne"	1979
110	"Saumur"	1989

Vineyard

name	country
"Chablis"	"France"
"Riessling"	"Germany"
"Beaujolais"	"France"
"Bordeaux"	"France"
"Almaden"	"USA"
"Bourgogne"	"France"
"Saumur"	"France"

Fig. 6. Example database.

fields within a tuple. However, POOL was one of the boundary conditions of the project, which made it impossible to exploit clever memory allocation schemes and main memory data structures, and to experiment with the tuple representation.

2) Recovery: The recovery mechanism of PRISMA is based on the two-phase commit protocol together with logging and checkpointing techniques per relation fragment (see Fig. 4). Each OFM that participates in an update transaction records on its local log file the transaction updates, the transaction precommit decision, and finally, the global abort or commit status. When a log grows too large, the OFM can decide locally to write a checkpoint file to disk, thereby clearing the log. After a system crash each OFM can recover independently by reloading the most recent checkpoint from disk and replaying the update statements of committed transactions from the log file. Note, that the PRISMA architecture is designed to make use of parallel logging [1] and recovery to reduce the overhead of disk I/O.

In some cases, it is possible that the OFM was in a precommit state at the moment of the crash. Then the recovery mechanism of the OFM must find out the state of the global transaction at the time of the crash. This information is kept up to date in a global *transaction log* by the TM during transaction processing. The transaction state can be *active*, *committed*, or *aborted*. At recovery time, the OFM retrieves the transaction state from the *transaction log*. If the state is *aborted* or *active*, the OFM will not replay the update statements of the last transaction on the log.

The database is protected against media failures by the stable file system of the POOMA system. This file system employs a file replication technique that keeps a copy of each file on a different disk. After a media failure, the POOMA system software is responsible for bringing the file system back into a consistent state.

IV. Query Execution: An Example

To illustrate the dynamic aspects of the DBMS architecture, the execution of an example query is described. The database in Fig. 6 is used in this query (this example is borrowed from [19]). The relations are fragmented on their first attribute. Person and Drinks are fragmented into two fragments (Person1, Person2, Drink1, Drink2), and Wine into three fragments (Wine1, Wine2, Wine3); Vineyard has one fragment (Vineyard1). The horizontal lines in Fig. 6 indicate the fragment boundaries. The first attributes of Wine, Person, and Vineyard are unique keys. The domain

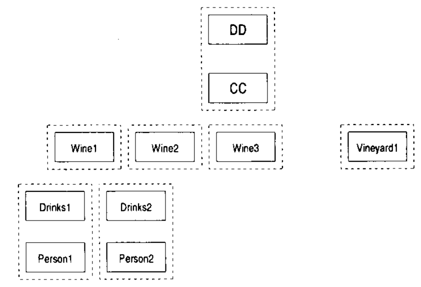


Fig. 7. PRISMA/DB filled with the example database.

of the age field of the Person relation is restricted to integers in the interval [0, 120]]. Furthermore, there are the obvious referential integrity constraints in this schema: from Drinks.pers to Person.id, from Drinks.wine to Wine.id, and from Wine.name to Vineyard.name. The corresponding fragments of Person and Drinks reside on the same processor; all other fragments have a private processor. Fig. 7 shows PRISMA/DB with this database stored in it, when idle. In this figure, the OFM-components are labeled with the name of the fragment they store, instead of the label "OFM" as in Fig. 2. The dotted boxes in Fig. 7 represent processors.

The physical data organization of the example database illustrates the flexibility of the data storage system: an arbitrary number of fragments is possible for each relation, and each fragment can be allocated to any processor that has enough memory space to hold the data.

A. A Retrieval Query

We now assume that the database from Fig. 6 is stored in PRISMA/DB. As an example of a retrieval query, we will find the names of the persons that drink German wine. SQL is used as query language.

```
SELECT Person.name FROM
Person, Drinks, Wine, Vineyard
WHERE Person.id = Drinks.person AND
        Wine.id = Drinks.wine AND
        Wine.name = Vineyard.name AND
        Vineyard.country = "Germany."
```

To execute this query, an SQL compiler is created. This compiler checks the syntactic and semantic correctness of the query. To do the semantic checking, the SQL compiler contacts the DD, that supplies information about the schema of the relations that are used in a query. If the query is found correct, it is translated into XRA-R:

```
<*2*> select (11="Germany" and 1=4 and 5=7
              and 8=10,
              cp(Person, Drinks, Wine, Vineyard)).
```

In this XRA construct, numbers refer to attributes, the keyword cp is the Cartesian product (in the Cartesian product, the attributes of the operands are concatenated, so the result has 11 attributes), and < *2* > indicates that the result of the operation is to be projected on its second attribute. This XRA-R query is handed to the QO. The QO compiles the query into XRA-F and optimizes it. Just simple optimizations are used in the current version of PRISMA/DB: selections and projections are pushed down as far as they are possible and useful, and joins are distributed over unions. No proper algorithm to decide on the join order, and on the degree of parallelism for each join is implemented yet (see Section VI-A). The QO contacts the DD to get fragmentation information for the relations in the query. Also, the DD can supply statistics about relations and fragments to the QO. A possible resulting XRA-F query is:

```
c1 = Person1
c2 = Person2

{c3,c4,c5} =(5) <*2,5*> join (Drinks1, 1=4, c1)
{c6,c7,c8} =(5) <*2,5*> join (Drinks2, 1=4, c2)

{c9,c10} =(4) <*1,4*> join ({c3,c6}, 2=3, Wine1)
{c11,c12} =(4) <*1,4*> join ({c4,c7}, 2=3, Wine2)
{c13,c14} =(4) <*1,4*> join ({c5,c8}, 2=3, Wine3)

{c15,c16}" =(1) <*1*> select (2 = "Germany,"
                              Vineyard1)

c17 = <*1*> join ({c9,c11,c13}, 1=3, c15)
c18 = < *1*> join ({c10,c12,c14}, 1=3, c16)

?union (c17, c18).
```

This program looks pretty complex, however, the corresponding execution infrastructure in Fig. 8 illustrates its meaning. The facilities of XRA that are explained in Section III—B are used in this program:

$\{ca, cb\}$ as operand refers to an operand that consists of multiple streams of input data.

$\{ca, cb\} = (x)$ indicates that the result of an operation has to be split on attribute x over multiple output streams.

$< *a, b* >$ indicates that the result of an operation has to be projected on attributes a and b.

Person can be joined to Drinks without refragmentation, because these relations are fragmented on the join attribute. The Person fragments have to be sent to the Drink fragments, however, because they are managed by other OFM's. The results of these joins have to be redistributed to join them to Wine. Finally, the result of the selection from Vineyard, and the result of the join to Wine are redistributed to calculate their join on two processors. The results are united and sent to the user. Before the tuples are sent of-node, they are projected on the relevant attributes to reduce the communication costs.

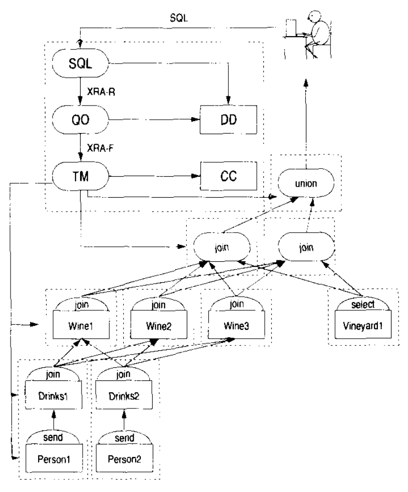


Fig. 8. PRISMA/DB executing the example query.

The XRA-F program is handed to the TM, which creates the execution infrastructure and coordinates the execution. The necessary execution infrastructure is shown in Fig. 8. For each fragment that is used, the TM asks an S-lock from the CC. When the lock is acquired the fragment can be accessed. As explained in Section III-E, an LTM has to be attached to an OFM to execute relational operations on base fragments (in the figure these LTM's are represented by half ovals on top of each OFM). Operations that do not have any base fragment as an operand are executed by independent LTM's (ovals in the figure). The TM creates all LTM's and initializes them with the XRA statement they have to execute. The (half) ovals in the figure are labeled with the XRA-statement they execute.

After its setup, the infrastructure is completely self-scheduling. Each LTM connects to its destination(s) (references to them are incorporated in the XRA statement that is executed); as each LTM works independently, this coordination phase is intrinsically parallel. As soon as an LTM has connected to all its destinations, it can start processing the available data. Base data are directly available, but data that is coming in via channels may have to be waited for. The infrastructure in execution works like an assembly line, with the LTM's as workers, and the data flowing along them. The LTM's are activated by the data that are available. Each operation terminates as soon as all operands have terminated. An operand terminates when as many EOF tuples have been encountered as there are channels in the operand. The entire query is ready when two EOF tuples have reached the final union. When ready, an LTM sends a ready message to the coordinating TM, which can shut down the execution when all participants are ready. After the commit of a transaction, the locks are released, and all LTM's with their data and the TM are discarded. It appears that Peter, Paul, and Carel have drunk German wine. The example query execution shows all forms of intraquery parallelism.

- Each join (on the relation level) is executed in parallel (with degrees of 2, 3, and 2, respectively (intraoperation parallelism)).

- The join on Drinks and the join on Wines are executed parallel with the selection on Vineyard (interoperation parallelism).
- The fact that PRISMA/DB is a main memory DBMS allows us to use the pipelining join algorithm (see Sections III-E and VI-A). By using this algorithm, the three parallel join operations form a pipeline, in which all levels can execute at the same time (provided enough data are used of course). So, there we have interoperation pipelining. Another short pipeline starts from Vineyard.

The example query execution illustrates the flexibility of PRISMA/DB. As the fragmentation degree of the base relations, the degree of parallelism of each relational operation, and the allocation of OFM's and LTM's can be chosen freely, the systems allows experimentation with a very broad class of execution strategies.

B. Inserting a Tuple

As an example of an update query, we will show how a tuple is inserted into the database. A "Saumur" 1990, with id 111 is added to the database. The first phase of this transaction (the actual insertion) is equivalent to the first phase of the retrieval query: the SQL compiler generates an XRA-R insert statement:

```
insert(Wine, {[111, "Saumur", 1990]})
```

which is optimized into XRA-F by the QO:

```
insert(Wine2, {[111, "Saumur", 1990]}).
```

Note, that the QO has replaced the insert into a relation by an insert into one fragment of the relation, instead of into all fragments that belong to the relation. This optimization can be done for single-tuple inserts.

The TM again generates the execution infrastructure for this query, which in this case consists of one LTM, which is attached to Wine2.

The difference between retrieval and update queries is apparent at commit time: now the correctness of the transaction with respect to the integrity constraints is checked, and the update must be made permanent in the OFM.

Two referential integrity constraints are defined on relation Wine; one from Drinks to Wine, and one from Wine to Vineyard. The first constraint cannot be violated by an insert into Wine, but the second one can: it has to be checked whether a "Saumur" tuple exists in Vineyard. To check integrity constraints, compiled versions of these constraints are stored in the DD with the fragments. At commit time, the TM asks the DD for the constraints that have to be checked, when an insert into Wine2 has been executed. The DD returns an XRA program to the TM, and the TM executes this program before it actually commits the transaction. In this case, the returned XRA program looks as follows:

```
c1 = unique (<*2*>Wine2)
c2 = <*1*>Vineyard1
c3 = c1 - c2
alarm (c3).
```

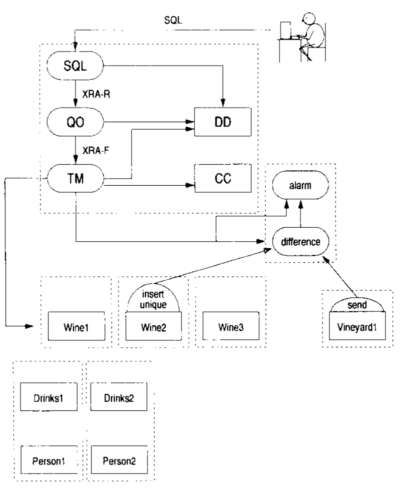


Fig. 9 PRISMA/DB executing an insert.

The alarm statement generates an abort, when the cardinality of its operand is greater than zero. The complete execution infrastructure that is built for the insert transaction is shown in Fig. 9. Note that the setup of the infrastructure for constraint enforcement is done parallel to the execution of the insert query.

When the execution of the insert and the constraint enforcement program is ready, the TM knows whether the transaction can commit or not. In case of an abort, an abort message is sent to all participating base LTM's. When the transaction can commit, the TM sends a precommit message to all participating base LTM's, which start making the insert permanent in the way described in Section III-F. A commit message ends the execution of the insert statement.

V. PERFORMANCE

As explained in the introduction, meaningful performance evaluation was only possible after the completion of the second version of PRISMA/DB. The results of the first performance tests in the spring of 1991 were bad due to synchronization problems in the system [47]. Some parts of the system were redesigned to eliminate these problems. The resulting version of the system was completed in the late fall of 1991. The performance evaluation of this system is described here.

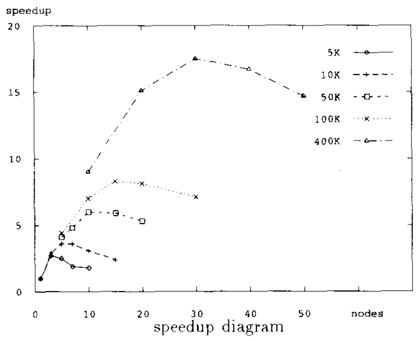
Some queries from the Wisconsin Benchmark are used to evaluate the performance [9]. This paper describes the most important aspects of the performance of PRISMA/DB as a main memory system. A full description of the performance can be found in [47].

A. Selection Queries

A query that selects 1% of its input is used to evaluate the performance of selection queries. The source relation is fragmented over a number processors and the selection criterion is not on the partitioning attribute, so all fragments have to be searched for qualifying tuples. The result is stored fragmented without redistribution on the processors generating

processors	5K	10K	50K	100K	400K
1	480	912			
3	**176**	306			
5	188	**248**	775	1416	
7	208	252	656		
10	262	292	**524**	876	2796
15		384	530	**735**	
20			596	760	1646
30				860	**1426**
40					1486
50					1692

response times in ms



speedup diagram

Fig. 10. Performance of selection queries.

result tuples (as PRISMA/DB is a main memory system, results are not written on disk). Different sizes for the source relation are used, ranging from 5000 (5K) tuples to 400 000 (400K) tuples. For each source relation size, a speedup experiment is done. The numbers of processors used are adjusted to the size of the source relation, using larger numbers of processors for larger source relations.

Fig. 10 shows the response times resulting from the selection queries, and the speedup diagrams that can be calculated from them. All response times are given in milliseconds. The **best** response time for each source relation size is printed in bold type.

The response times are a measure for the absolute performance of the system. The absolute performance figures are reasonable compared to other systems. Comparison of the absolute performance of systems is hard, because there are too many differences between systems in hardware, functionality, etc. However, to give an indication, Fig. 11 lists the response times of some other systems, with the number of processors used for a 1% selection from 100K tuples. The absolute performance of PRISMA/DB seems reasonable from these data. However, as PRISMA/DB is a main memory system, it should outperform all disk-based systems mentioned in Fig. 11. This issue is discussed after the presentation of the other performance results.

The speedup characteristics illustrate the relative performance of the system. Linear speedup is the ultimate goal for parallel processing. However, a system that uses sequential initialization of the subtasks in the parallel execution of an operation can only get linear speedup for small numbers of processors. Our performance measurements show this phenomenon; we will now explain why. The response time to a query consists of two components.

- The TM sequentially creates and initiates the participating LTM's. This yields a component in the response time that is growing linearly with the number of processors.
- Each LTM has to do a certain amount of local processing. This yields a component in the response time that is inversely proportional to the number of processors.

This simple reasoning leads to the observation that adding more processors to a parallel task ultimately degrades the performance in any system that uses sequential initialization

name	processors	response time	
Teradata	20	28 220	[16]
GAMMA(VAX)	8	13 830	[16]
Silicon DBM	3	10 900	[32]
PRISMA/DB	15	735	
GAMMA(Intel)	30	150	[15]

Fig. 11. Response times of some parallel DBMS's to a 1% selection from 100 tuples in ms.

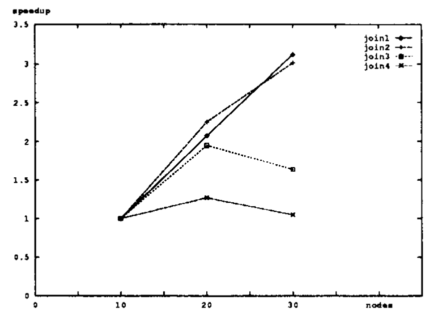
of these tasks. However, it is possible that the degrading performance is not measured, because it occurs only for larger numbers of processors than available. PRISMA/DB does yield degrading performance for a number of processors that is lower than the number of available processors and the reason for this is twofold: firstly, relatively small source relations are taken into account, which leads to a small amount of local processing. The speedup diagrams show that the optimal number of processors indeed is lower for smaller source relations. The second reason is the main memory nature of the system. The sequential component in the response time consists of a lot of coordination and thus message passing. Therefore, this component does not benefit from the main memory nature of the system. The costs of the local processing, however, are lowered by the system being main memory. Therefore, the optimal number of processors to be used for a parallel task is lower on a main memory system than on an equivalent disk-based system. In other words, we can say, that in a main memory system, the local processing is so fast that it is hard to start all parallel components before the first one is ready. A more extensive and formal coverage of this issue can be found in [47].

The observation about the behavior of a parallel main memory system has implications for the hardware configuration that should be chosen for such a system. Obviously, a main memory system needs a large amount of primary memory. However, as the maximal size of a subtask in a parallel task is directly related to the size of the memory of one processor, the amount of memory per processor should be fairly large to yield subtasks that are large enough to allow performance gain from parallelism.

In the next section, the parallel execution of join queries is discussed. Because join queries are more expensive than selections, their speedup characteristics are expected to be better.

#	join1	join2	join3	join4
10	6180	6132	6324	9036
20	2980	2718	3240	7100
30	1978	2034	3838	8566

response times in ms



speedup characteristics

Fig. 12. Performance of join queries.

B. Join Queries

The join query used in the performance experiments is a query joining a 10K tuple relation to a 100K tuple relation in which every tuple of the 10K relation matches to one tuple in the 100K relation, so the result consists of 10K tuples. This query is called the joinABprime query in [9]; A is the 100K relation and Bprime is the 10K relation. Four different execution strategies were tested, which are called join1 through join4 in the sequel:

join1: The relations are initially fragmented on the join attribute into equal numbers of fragments, and the corresponding fragments reside on the same processor.

join2: The relations are fragmented in the same way as for join1, but all fragments reside on different processors. The Bprime fragments are sent to the A fragments for joining.

join3: Relation A is fragmented on the join attribute and relation Bprime is fragmented on another attribute into equal numbers of fragments. All fragments reside on different processors. Relation Bprime is redistributed and sent to relation A for joining.

join4: Both relations are fragmented on another attribute than the join attribute into equal numbers of processors. All fragments reside on different processors. Both relations are redistributed and sent to the join processors for joining.

These four strategies were tested using 10, 20, and 30 processors for the joins combined with a fragmentation degree of 10, 20, or 30 for the initial fragmentation of the relations.

Fig. 12 shows the response times measured in this experiment, and the speedup with respect to the response time of the 10-processor queries. Note, that in this case, linear speedup yields a speedup factor 3 for the 30-processor queries.

The achieved absolute performance for "join1" and "join2" is good compared to other systems. Fig. 13 lists the response times for the same query reported by other projects. Again, it is hard to compare systems, as they differ in many ways. Yet, we like to report that the response time measured on PRISMA/DB outperforms all other reported performance figures on this query.

Join1 and join2 show, apart from a good absolute performance, good speedup characteristics. The speedup is even slightly superlinear. This is caused by some synchronization problems for the queries using 10 processors.

name	processors	response time	
Teradata	20	131300	[16]
GAMMA(VAX)	8	15600	[16]
Silicon DBM	3	23900	[32]
HC16-186	16	10000	[12]
GAMMA(Intel)	30	3340	[15]
PRISMA/DB	30	1978	

Fig. 13. Response times of some parallel DBMS's to a 100K by 10K join, fragmented on the join attribute.

The speedup characteristics of join3 are disappointing, and join4 is even worse. The reason for this is as follows. Join3 and join4 need redistribution of the operands. This redistribution is expressed in XRA, and the expression for it is large, and grows for larger degrees of parallelism, as the number of destinations grows with the degree of parallelism. The TM sequentially sends the same large expression to each LTM, and because POOL-X does not support broadcasting, the overhead for sending an XRA-expression off-node is made for each fragment. Join3 needs to go through the redistribution of only one operand, but join4 has to redistribute both operands making things even worse.

Here we are at a point where we have to pay for both forms of flexibility offered by the system. Firstly, using POOL-X facilitated the development of a flexible architecture, but the high level interface offered by POOL-X makes it impossible to solve the problem of sending large XRA-expressions to many LTM's. Secondly, XRA was developed to express a wide variety of parallel execution plans, but the expressions that are generated in plans that have a high degree of parallelism grow larger than we want.

Although there are some problems, we feel that PRISMA/DB with the performance reported in this section, offers a very good platform to experiment with parallel query execution, especially to study the execution of complex queries, in which the degree of intraoperator parallelism does not need to be very large.

C. Concluding Remark

This section discussed the performance of PRISMA/DB. Both the relative and the absolute performance were discussed. The discussion of the relative performance yielded useful results for the implementation of parallel main memory systems. The absolute performance, however, is not as good as might be expected from a main memory system. Specifically, a main memory system should outperform a disk-based system by at least an order of magnitude. PRISMA/DB does not achieve this and the reason is two-fold. Firstly, the hardware used is (although state-of-art when the PRISMA project started) outdated now. Also, due to the use of memory extension boards, the hardware does not run full-speed (see Section II). Secondly, the use of an experimental high level programming language has performance penalties: the high level of the language makes low level optimizations hard or impossible, and also, the new compiler was not optimized in detail. However, the use of the high level programming platform was profitable too, in the sense that we did not have to bother about all sorts of nasty low level details, so that a fully functional DBMS could be finished within a reasonable time frame.

VI. Current Research

This section describes how the flexibility of PRISMA/DB is used in our research on parallel query execution and on integrity constraint enforcement.

A. Multijoin Queries

The flexibility of the query execution layer of PRISMA/DB is used to study the parallel execution of complex queries. A complex query is a query that consists of multiple relational operations. Multijoin queries are used as an example of complex queries in this paper. Important questions consist of the following.

- What is the best join order in a parallel environment?
- What degree of parallelism should be used for each join operation?
- How to allocate processors to each join operation?
- How does the initial data distribution influence the query execution?

References [43]–[46], [48] are reports on this research. In those papers, the pipelining hash join algorithm (see Section III-E) is introduced as an algorithm that has fewer constraints on the order in which operand tuples can be processed than the known hash join algorithms, and as such, it is expected to yield significant performance gain from interjoin pipelining. Its behavior in linear and bushy query plans for a restricted class of multijoin queries was studied, using simulation and analytic mathematical analysis (the distinction between left-deep and right-deep linear plans [34] does not exist here, because the pipelining hash join is a symmetric algorithm). Simulation was used, as at the time this research was started, the final version of PRISMA/DB was not ready yet. The results of the study show that effective parallelism can be achieved in a join pipeline. Also, it was shown that join queries with small operands are better off with a bushy query plan, and join queries with large operands prefer a linear schedule.

Currently, this research is continued as follows. Firstly, the operational PRISMA/DB prototype is used to confirm the results from [43], secondly, we want to extend the study to a broader class of multijoin queries, and finally, intraoperation parallelism for the individual join operations will be considered.

B. Integrity Control

One of the current research directions in the PRISMA context is integrity control in parallel main memory database systems. The main topics in this research are software architectures for integrity control, the effects of data distribution and parallel enforcement, and ways to improve the performance of integrity constraint enforcement in parallel environments. The emphasis on parallelism and performance in constraint enforcement contrasts this research to that performed in the context of other DBMS projects like SABRINA [35], POSTRGRES [41], and STARBURST [24].

In this research, the basic software architecture for integrity control is based on the transaction modification principle as explained in the section on transaction management. This principle enables the use of the standard query execution machinery for constraint enforcement and deals correctly with transaction serializability and atomicity requirements. As discussed in [21], the basic architecture can be extended in a number of ways to obtain a better performance of integrity control.

The effects of data distribution and parallel enforcement are described in detail in [20]. Here, attention is paid to the translation of constraints in a functional specification (first-order logic) to an operational specification in XRA, the removal of fragmentation transparency and the optimization of constraints in a parallel context, and to the mapping of constraints to the parallel query execution machinery of PRISMA/DB. The concepts can be used easily within the transaction modification context.

A performance evaluation of constraint enforcement on the PRISMA/DB prototype has lead to two important observations. In the first place, parallelism has proven to be a good way to deal with the high processing costs associated with constraint enforcement; transaction execution times including integrity control can be strongly improved by parallel execution. Secondly, the relative costs of constraint enforcement have shown to be quite acceptable in comparison to transaction execution without any integrity control; typical figures are a few percent for very simple constraints and about 100 % for referential integrity constraints in the worst case. The fact that PRISMA/DB uses main memory storage has a positive influence on these figures, since constraint enforcement is (mainly) a retrieval process, whereas update transactions require secondary storage operations.

Research is being performed on special-purpose communication protocols for constraint enforcement at the lower levels of PRISMA/DB. The main goal of these protocols is to decrease the control overhead imposed by the transaction management process in constraint enforcement. Further gains in performance can be expected from an optimal scheduling of constraint enforcement [22].

VII. Conclusions and Future Research

In this paper, we have discussed the design and implementation of PRISMA/DB, a parallel, main memory RDBMS. The design of the system can be characterized by two main ideas: use of parallelism and main memory data storage to provide high performance in query processing, and use of a high level object-oriented language to obtain a modular and flexible system architecture that can be used easily for experimentation with functionality and performance.

Currently, the second prototype of the DBMS, called PRISMA/DB1, is running on hardware configurations up to 100 nodes. The prototype provides complete DBMS functionality among which concurrency control, integrity control, and crash recover facilities. Extensions of the functionality can be added easily, like automatic loading and unloading mechanisms to be able to handle databases that do not fit into the main memory of the system. The absolute performance of the prototype has shown to be comparable to other state-of-the-art parallel database machines. The relative performance with respect to software and hardware configuration has led to new insight into the behavior of parallel main memory systems.

The choice of an experimental object-oriented implementation language for PRISMA/DB has had an important impact on the project. The language has proven to be a great advantage in obtaining a well-structured and flexible software architecture. The mapping of DBMS components onto active objects in this language enables a natural modularization of the system with clear interfaces. On the other hand, the choice of a high level implementation language has shown to be a drawback in obtaining optimal performance, since no explicit control over the hardware and low level processes is possible.

PRISMA/DB1 is used as an experimental platform for a number of research activities. In the first place, experiments with multioperation queries and parallel integrity control, as described in the previous section, will be conducted on the prototype. Furthermore, PRISMA/DB1 is used for the implementation of parallel algorithms for transitive closure operations [25]–[27]; this enables the parallel computation of recursive queries on PRISMA/DB1. Also, the system will be used as an experimental implementation platform for a NF2 layer that supports complex objects [38]; because flattening a complex database schema onto a relational schema yields a schema with many referential integrity constraints, and queries that need many join operations, this layer will rely heavily on the referential integrity control and parallel multijoin facilities of the system.

References

[1] R. Agrawal and D. J. DeWitt, "Recovery architectures for multiprocessor database machines," in *Proc. ACM-SIGMOD 1985 Int. Conf. on Management of Data*, Austin, TX, May 28–31, 1985.

[2] P. America, "POOL-T, A parallel object-oriented language," in *Oject Oriented Concurrent Programming*, A. Yonezawa and M. Tokoro, eds. Cambridge, MA: MIT, 1987, pp. 199–220.

[3] P. America, "Issues in the design of a parallel object-oriented language," *Formal Aspects Comput.*, vol. 1, pp. 366–411, 1989.

[4] ——, ed., *Proc. PRISMA Workshop on Parallel Database Systems*. New York: Springer-Verlag, 1991.

[5] P. M. G. Apers, M. A. W. Houtsma, and F. Brandse, "Processing recursive queries in relational algebra," in *Proc. Second IFIP 2.6 Working Conf. on Database Semantics*, Albufeira, Portugal, Nov. 3–7, 1986, pp. 17–39.

[6] C. A. van den Berg and M. L. Kersten., "Engineering a main memory DBMS," *CWI Quart.* Centre for Mathematics and Computer Science, Amsterdam, The Netherlands, 1991.

[7] C. A. van den Berg, M. L. Kersten, and K. Blom, "A comparison of scanning algorithms," in *Proc. Int. Conf. on Databases, Parallel Architectures and Their Applications*, Miami, Mar. 1990.

[8] B. Bergsten, M. Couprie, and P. Valduriez, "Prototyping DB3S, A shared-memory parallel database system," in *Proc. First Int. Conf. on Parallel and Distributed Information Systems*, Miami Beach, FL, Dec. 1991, pp. 226–235.

[9] D. Bitton, D. J. DeWitt, and C. Turbyfill, "Benchmarking database systems — A systematic approach," in *Proc. Ninth Int. Conf. on Very Large Data Bases*, Florence, Italy, Oct. 31–Nov. 2, 1983.

[10] H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez, "Prototyping Bubba, A highly parallel database system," *IEEE Trans. Knowl. Data Eng.* vol. 2, pp. 4–24, 1990.

[11] H. Boral and S. Redfield, "Database machine morphology," in *Proc Eleventh Int. Conf. on Very Large Data Bases*, Stockholm, Sweden, Aug. 21–23, 1985.

[12] K. Bratbergsengen and T. Gjelsvik, "The Development of the CROSS8 and HC16–186 (database) computers," in *Proc. Sixth Int. Workshop on Database Machines*, Deauville, France, June 1989, pp. 359–372.

[13] W. J. H. J. Bronnenberg, L. Nijman, E. A. M. Odijk and R. A. H. V. Twist, "DOOM: A decentralized object-oriented machine," in *IEEE Micro*, Oct. 1987.

[14] S. Ceri and G. Pelagatti, *Distributed Databases, Principles and Systems*. New York: McGraw-Hill, 1984.

[15] D. J. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. Hsiao, and R. Rasmussen, "The GAMMA database machine project," *IEEE Trans. Knowl. Data Eng*, vol. 2 pp. 44–62, Mar. 1990.

[16] D. J. DeWitt, S. Ghandeharizadeh, D. Schneider, R. Jauhari, M. Muralikrishna, and A. Sharma, "A single user evaluation of the GAMMA database machine," in *Proc. Fifth Int. Workshop on Database Machines*, Karuizawa, Japan, Oct. 1987.

[17] 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 1984 Int. Conf. on Management of Data*, Boston, MA, June 18–21, 1984, pp. 1–8.

[18] M. Eich, "A classification and comparison of main memory database recovery techniques," in *Proc. 1987 Database Engineering Conf.*, 1987, pp. 332–339.

[19] G. Gardarin and P. Valduriez, *Relational Databases and Knowledge Bases*. Reading, MA: Addison-Wesley, 1989.

[20] P. W. P. J. Grefen and P. M. G. Apers, "Parallel handling of integrity constraints on fragmented relations," in *Proc. Second Int. Symp. on Databases in Parallel and Distributed Systems*, Dublin, Ireland, July 2–4 1990, pp. 138–145.

[21] ——, "Integrity constraint enforcement through transaction modification," in *Proc. 2nd Int. Conf. on Database and Expert Systems Applications*, Berlin, Germany, July 1991.

[22] ——, "Dynamic action scheduling in a parallel database system," in *Proc. Conf. on Parallel Architectures and Languages in Europe*, Paris, France, 1992.

[23] P. W. P. J. Grefen, A. N. Wilschut, and J. Flokstra, "PRISMA/DB1 user manual," Memo. INF9106, Universiteit Twente, Enschede, The Netherlands, 1991.

[24] L. Haas, J. C. Freytag, G. Lohman, and H. Pirahesh, "Extensible query processing in Starburst," in *Proc. ACM-SIGMOD 1989 Int. Conf. on Management of Data*, Portland, OR, May 31–June 2, 1989.

[25] M. A. W. Houtsma, P. M. G. Apers, and S. Ceri, "Distributed transitive closure computations: The disconnection set' approach," in *Proc. Sixteenth Int. Con. on Very Large Data Bases*, Brisbane, Australia, Aug. 13–16, 1990, pp. 335–346.

[26] M. A. W Houtsma, F. Cacace, and S. Ceri, "Parallel hierarchical evaluation of transitive closure queries," in *Proc. First Int. Conf. on Parallel and Distributed Information Systems*, Miami Beach, FL, Dec. 1991.

[27] M. A. W. Houtsma, A. N. Wilschut, and J. Flokstra, "Implementation and performance evaluation of a parallel transitive closure algorithm on PRISMA/DB," Memo. INF92–45, Universiteit Twente, Enschede, The Netherlands, July 1992.

[28] M. L. Kersten, "Using logarithmic code-expansion to speedup index access," in *Foundations of Data Organization and Algorithms* New York: Springer-Verlag, June 1989, pp. 228–232.

[29] E. van Kuijk, "Semantic query optimization in distributed database systems," Ph.D. dissertation, Univ> of Twente, 1991.

[30] T. J. Lehman and M. J. Carey, "Query processing in main memory

database management systems," in *Proc. ACM-SIGMOD 1986 Int. Conf. on Management of Data*, Washington, DC, May 28–30, 1986,pp. 239–250.

[31] ——, "A recovery algorithm for a high-performance memory-resident database system," in *Proc. ACM-SIGMOD 1987 Int. Conf. on Management of Data*, San Francisco, CA, May 27–29, 1987.

[32] M. D. P. Leland and W. D. Roome, "The silicon database machine: rational, design, and results," in *Proc. Fifth Int. Workshop on Database Machines*, Karuizawa, Japan, Oct. 1987.

[33] H. Garcia Molina, R. J. Lipton, and J. Valdes, "A massive memory machine," *IEEE Trans. Comput.*, vol. C-33, pp. 391–399, May 1984.

[34] D. A. Schneider and D. J. DeWitt, "Tradeoffs in processing complex join queries via hashing in multiprocessor database machines," in *Proc. Sixteenth Int. Conf. on Very Large Data Bases*, Brisbane, Australia, Aug. 13–16, 1990, pp. 469–480.

[35] E. Simon and P. Valduriez, "Design and implementation of an extendible integrity subsystem," in *Proc. ACM-SIGMOD 1984 Int. Conf. on Management of Data*, Boston, MA, June 18–21, 1984.

[36] C. J. Skelton, C. Hammer, M. Lopez, M. J. Reeve, P. Townsend, and K. F. Wong, "EDS: A parallel computer system for advanced information processing," in *Proc. Parallel Architectures and Languages in Europe*, Paris, France, June 1992, pp. 877–892.

[37] J. van der Spek, "POOL-X and its implementation," in *Proc. PRISMA Workshop on Parallel Database Systems*, Noordwijk, The Netherlands, 1990, pp. 309–344.

[38] H. J. Steenhagen and P. M. G. Apers, "ADL — An algebraic database language," in *Proc. Computing Science in the Netherlands*, Utrecht, The Netherlands, Nov. 1990, pp. 427–442.

[39] M. Stonebraker, "Implementation of integrity constraints and views by query modification," in *Proc. ACM-SIGMOD 1975 Int. Conf. on Management of Data*, San Jose, 1975.

[40] M. Stonebraker, R. Katz, D. Patterson, and J. Ousterhout, "The design of XPRS," in *Proc. Fourteenth Int. Conf. on Very Large Data Bases*, Los Angeles, CA, Aug. 29–Sept. 1, 1988.

[41] M. Stonebraker, L. A. Rowe, and M. Hirohama, "The implementation of POSTGRES," *IEEE Trans. Knowl. Data Eng.* vol. 2, Mar. 1990.

[42] P. Watson and P. Townsend, "The EDS parallel relational database system," in *Proc. PRISMA Workshop on Parallel Database Systems*, Noordwijk, The Netherlands, 1990.

[43] A. N. Wilschut and P. M. G. Apers, "Dataflow query execution in a parallel main memory environment," in *Proc. First Int. Conf. on Parallel and Distributed Information Syst.*, Miami Beach, FL, Dec. 1991.

[44] ——, "Dataflow query execution in a parallel main memory environment," to be published.

[45] ——, "Pipelining in query execution," in *Proc. Int. Conf. on Databases, Parallel Architectures and their Applications*, Miami, Mar. 1990.

[46] A. N. Wilschut, P. M. G. Apers, and J. Flokstra, "Parallel query execution in PRISMA/DB," in *Proc PRISMA Workshop on Parallel Database Systems*, Noordwijk, The Netherlands, Sept. 1990.

[47] A. N. Wilschut, J. Flokstra, and P. M. G. Apers, "Parallelism in a main memory system: The performance of PRISMA/DB," in *Proc. 18th Int. Conf. on Very Large Data Bases*, Vancouver, Canada, Aug. 23–27, 1992.

[48] A. N. Wilschut and S. A. van Gils, "A model for pipelined query execution," Memo. INF91-34, Univ. Twente, Enschede, The Netherlands, 1991.

[49] A. N. Wilschut, P. W. P. J. Grefen, P. M. G. Apers, and M. L. Kersten, "Implementing PRISMA/DB in an OOPL," in *Proc. Sixth Int. Workshop on Database Machines*, Deauville, France, June 1989, pp. 359–372.

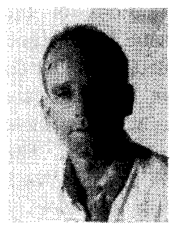
Peter M. G. Apers (M'84) received the Ph.D. degree from Vrije University, Amsterdam, The Netherlands, in 1982.

He has worked as a Researcher with the University of California at Santa Cruz and Stanford University. He is currently a Full Professor with the University of Twente, The Netherlands, where he leads a group working on object-oriented data models, complex object databases, optimization of logical query languages, parallelism in database management systems, and database machines.

Dr. Apers is a member of ACM. He currently serves on the editorial boards of *Data and Knowledge Engineering* and *Distributed and Parallel Database Systems.*

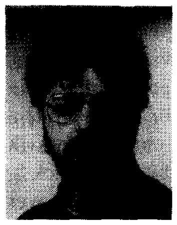
Carel van den Berg received the M.S. degree in computer science from the University of Amsterdam, The Netherlands, in 1986. He is currently working toward the Ph.D. degree.

Since 1987, he has been working for CWI and until 1990, he participated on the database machine design and implementation for the PRISMA project. His research interests include object-oriented database systems, performance analysis, and parallel and adaptive systems.



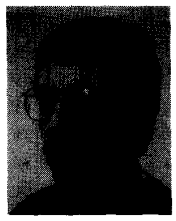
Jan Flokstra received the B.S. degree in computer science from the HIO, Enschede, The Netherlands, in 1986.

Since 1986, he has been a Research Programmer with the University of Twente, The Netherlands, where he has worked on the design and implementation of PRISMA. Currently, he is involved in the implementation of the object-oriented database specification language TM.



Paul W. P. J. Grefen received the M.S. degree in computer science and the Ph.D. degree from the University of Twente, The Netherlands, in 1986 and 1992, respectively.

In 1987, he joined the PRISMA project, where he mainly worked in the field of transaction management and integrity control. He is currently an Assistant Professor with the University of Twente.



Martin L. Kersten received the Ph.D. degree in computer science from the Vrije University, Amsterdam, The Netherlands, in 1985.

In 1985, he joined CWI, The Netherlands, where he set up the Database Research Group. From 1986 to 1990, he was the Co-Designer of the PRISMA database machine. Int the follow-up ESPRIT-II project TROPICS, he was responsible for the development of an enhanced version of SQL to cater with documents and geographical data. Since 1989, he has been leading a national project on the exploitation of the Amoeba distributed system for advanced database management. He is currently Head of the Computer Science Department, CWI. He is also an Associate Professor with Vrije University, where he teaches advanced courses on database technology. His current research interests are database programming languages, distributed and parallel object-oriented database systems, dynamic query optimization, and performance assessment of database systems. He is the author or co-author of over 60 technical papers.

Dr. Kersten is a member of ACM.

Annita N. Wilschut received the B.S. and M.S. degrees in biology from Vrije University, The Netherlands, in 1976 and 1980. She is currently working toward the Ph.D. degree.

In 1985, she joined the Case Center for Computer-Aided Design, Michigan State University as a Research Assistant. In 1986, she worked as a Research Fellow with the Andy Tanenbaum's Group, Vrije University, on distributed operating systems. In 1987, she joined the PRISMA group, University of Twente, where she mainly worked on parallel query execution.