PRISMA/DB: A Parallel, Main-Memory Relational DBMS

Peter M. G. Apers[†] Carel A. van den Berg[‡] Jan Flokstra[†]
Paul W. P. J. Grefen[†] Martin L. Kersten[‡] Annita N. Wilschut[†]

† University of Twente
P.O.Box 217, 7500 AE Enschede, the Netherlands
Phone: +3153-893705
apers@cs.utwente.nl flokstra@cs.utwente.nl grefen@cs.utwente.nl annita@cs.utwente.nl

‡ Center for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, the Netherlands
Phone: +3120-5924054
carel@cwi.nl mk@cwi.nl

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 multi-processor. The achieved flexibility of the system makes it a valuable platform for research in various directions.

Keywords: Parallel, main-memory, relational database management system, design and implementation, architecture, query execution, experimentation, integrity constraints.

1 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 [Ame91] (of which PRISMA/DB is a subproject) is the design and realization of parallel hardware and software to implement the parallel object-orients programming language POOL, and the implementation of a non-trivial 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 GigaBytes 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 also 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 get a performance comparable to currently available prototype database machines. This performance has to be obtained by both parallelism (100 nodes) and main memory (16 Megabytes 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. 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 [WGA89].

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 eighties were predicted correctly in [GLH83]. Main memory in 1992 costs about $100K per Gigabyte. The potential benefits and problems of a MMDBMS were given in [DKO84] and a single prototype implementation of a shared-store MMDBMS was developed [LeR87].
During the project's life cycle an increasing number of papers appeared that address technical issues

[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

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 [Eic87,LeC87], parallelism in large-scale comparable (disk-based) systems, such as GAMMA [DGS90], Bubba [BAC90], and HC16-186 [BrG89]. The role of main-memory to hold the entire database is getting more support, as illustrated by the shared-store systems XPRS [SKP88], DBS3 [BCV91] and distributed store system EDS [WaT91].

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. Half way 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 get a better performance. Currently a 100K $\times$ 10K join of the Wisconsin benchmark runs in 2 seconds.

Research is now focused on a few topics to investigate the performance and the flexibility of the architecture: performance evaluation, parallel join evaluation, and parallel constraint enforcement, each of which will be discussed in more detail.

This paper is organized as follows. The next section briefly introduces the 100-node parallel multi-processor that is used, and the implementation language POOL-X. Section 3, first gives an overview of the DBMS architecture, and than 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 4 illustrates the dynamic aspects of the architecture via the description of an example query execution. Section 5 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 6 briefly describes the current research in the context of PRISMA/DB, and Section 7 summarizes and concludes the paper.

2 Hardware and software support

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

2.1 The POOMA machine

The POOMA machine is a shared-nothing, parallel multi-processor, which consists of 100 nodes. [BNO87] describes its design and the rational behind it in detail. Figure 1 shows the hardware configuration. Each node consists of a 68020 data processor with 16 Mbytes of memory, a disk, and a communication processor that links it to 4 other nodes using bidirectional links. 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; Figure 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.

2.2 POOL: A parallel object-oriented language

The programming language POOL-X [Ame87,Ame89,Spe91] is implemented on the POOMA machine, and is used as 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

Figure 1: Hardware configuration of the POOMA machine

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 run-time 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 inter-processor 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 half-way 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.

3 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

Figure 2: Global architecture PRISMA/DB

architecture are discussed in detail: the internal relational language 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 4, where examples of query execution are described in detail.

3.1 Overview

Figure 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 [WGA89]. As a result, the flexibility in the system architecture allows experiments with functionality.

The rectangles in Figure 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 short below.

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 data dictionary is the central storage of all meta-data of the system, like relation and constraint definitions, fragmentation information, and statistics. The concurrency controller controls concurrent access to the database. It uses a standard two-phase locking protocol with shared and exclusive locks. Further, 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 query language compiler 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 3.2). This component offers full fragmentation and allocation transparency to the user [CeP84]. Four different QLCs are available: a standard SQL interface, a logical query interface called PRISMAlog, that allows recursive queries [AHB88], 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 3.3). The QLCs and the QOs contact the DD to get the schema information and statistics needed for the tarnslation 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. Further, 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 3.4.

The data storage and query execution layer consists of the *one fragment managers* (OFMs) and the *local transactions managers* (LTMs). OFMs are permanent; they store and manage one fragment of a relation in the database. LTMs are transient; they are the relational engines in the system. The query execution layer is described in more detail in Section 3.5

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 inter-query parallelism on the query preprocessing level. Finally, allocation of OFMs and LTMs to different processors leads to parallel query execution in several froms. This issue is described in Section 3.3.

The main interface language between the various components of PRISMA/DB is formed by an extension to the relational algebra, called XRA [GWF91]. This language provides flexible, high level communication between the various query processing layers of the system. The language is discussed in detail below.

3.2 XRA

An Extended Relational Algebra (XRA) is used as internal representation of queries in the system. A full description of its syntax and semantics can be found in [GWF91]; 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-based 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 4.

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 4.

3.3 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, that consists of the QLCs 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.

Parallelism

In PRISMA/DB, various forms of parallelism can be used to speed up query execution. The standard terminology for parallelism [BoR85,WGA89] is used. To be complete, the used terminology is summarized here. Multiple users can use the system concurrently, yielding *inter-query* parallelism between their queries. Within a query, *intra-query parallelism* can be subdivided into *inter-operator*, and *intra-operator* parallelism. Orthogonal to this distinction, *pipelining* can be contrasted to (pure) *horizontal parallelism.* The term parallelism is often used as a synomym of horizontal parallelism. This paper adopts to this habit if no confusion is possible. Horizontal intra-operator 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.

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 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 the number of fragments used for the relation 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 scheme 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 run-time and the fragments can be reallocated to other processors. This allows experimentation with different allocation and fragmentation schemes in one session.

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

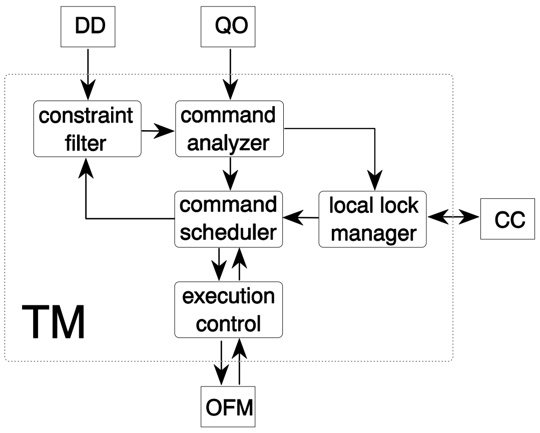


Figure 3: Architecture of the Transaction Manager

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 6.1) 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 [Kui91], 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 not a research issue currently.

3.4 Transaction Management and Integrity Control

The PRISMA/DB transaction manager (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 One Fragment Managers 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 Figure 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 lock manager. This module decides whether locks are

Figure 4: Local Transaction Management and its environment

already owned by the transaction, or have to be requested from the Concurrency Controller. 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 [Gre92]. 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 OFMs 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 OFMs. At transaction commit time, the integrity constraints to be enforced are retrieved from the Data Dictionary. 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 Data Dictionary, and can thus simply be appended to the end of the transaction, according to the transaction modification principle[2] [GrA91]. 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 OFMs involved in the transaction act as participants.

3.5 Query Processing

This section describes the query execution layer of PRISMA/DB. This layer consists of the OFMs, which store and manage base data, and the LTMs which are the relational engines of the systems. Figure 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

[2]Note the difference with the *query modification* approach [Sto75], where the selection predicates of updates are extended with the negation of constraint predicates.

allocation schemes for the fragments of the stored database.

An LTM is a transient object; it can execute relational operations. Typically, an LTM is created for each fragment operation in a query. Some LTMs are attached to an OFM, in which case they can directly access the fragment that is managed by that OFM; other LTMs 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 LTMs, or the stored result of a previous operation of the same LTM. LTMs 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 Figure 4, three LTMs 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 operation. 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, [BeK91,BKB90] 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 LTMs allows both pipelining and horizontal parallelism between different LTMs. In PRISMA/DB, we want to study both forms of inter-operation parallelism in the context of a main-memory system. In [WiA90] it is shown how special main-memory algorithms can be used that enhance the effective parallelism from pipelining. These pipelining algorithms aim at producing output as early as possible, so that a consumer of the result can start its operation. In particular, [WiA90] proposes a pipelining Hash-Join algorithm. This symmetric algorithm builds a hash-table for both operands, and it can produce a result tuple as soon as two matching tuples have reached the join-LTM. Where possible, pipelining algorithms are used for the implementation of relational operations.

3.6 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. Further, 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.

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 Figure 4). In particular, tuple layout, index creation, storage preservation and temporary storage are important for their design. See for instance [BeK91,Ker89,LeC86] 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 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.

Recovery

The recovery mechanism of PRISMA is based on the two-phase commit protocol together with logging and checkpointing techniques per relation fragment (see Figure 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 [AgD85] 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 Transaction Manager 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.

4 Query execution: an example

To illustrate the dynamic aspects of the DBMS architecture, the execution of an example query is described. The database in Figure 5 is used in this query (this example is borrowed from [GaV89]). 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 Figure 5 indicate the fragment boundaries. The first attributes of Wine, Person, and Vineyard are unique keys. The domain 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. Figure 6 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 Figure 2. The dotted boxes in Figure 6 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.

Figure 6: PRISMA/DB filled with the example database

4.1 A retrieval query

We now assume that the database from Figure 5 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
WHERE Person.id = Drinks.person  AND
      Wine.id = Drinks.wine AND
      Wine.name = Vineyard.name AND
      Vinyard.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),

Figure 7: PRISMA/DB executing the example query

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 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 6.1). 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 Figure 7 illustrates its meaning. The facilities of XRA that are explained in Section 3.2 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 OFMs. 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 tuples are sent of-node, they are projected on the relevant attributes to reduce the communication costs.

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 Figure 7. 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 3.5 an LTM has to be attached to an OFM to execute relational operations on base-fragments (in the figure these LTMs are represented by half ovals on top of each OFM). Operations that do not have any base-fragment as operand are executed by independent LTMs (ovals in the figure). The TM creates all LTMs 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 intrinsicly 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 LTMs as workers, and the data flowing along them. The LTMs are activated by the data being 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 LTMs 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 intra-query parallelism.

- Each join (on the relation level) is executed in parallel (with degrees resp. 2, 3, and 2 (intra-operation parallelism).
- The join on Drinks and the join on Wines are executed parallel with the selection on Vineyard (inter-operation parallelism).

Figure 8: PRISMA/DB executing an insert

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, that 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)
```

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 Figure 8. 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-LTMs. When the transaction can commit, the TM sends a precommit message to all participating base-LTMs, that start making the insert permanent in the way described in Section 3.6. A commit message ends the execution of the insert statement.

5 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 [Wil92]. 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 [BDT83]. 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 [Wil92].

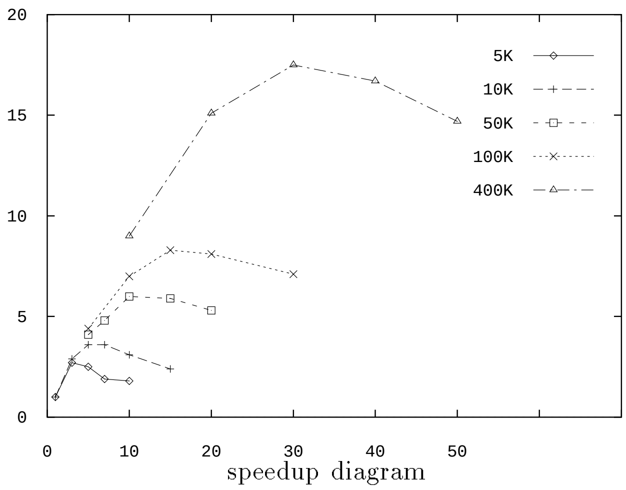
5.1 Selection queries

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 searched for qualifying tuples. The result is stored fragmented without redistribution on the processors generating 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 5 000 (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.

Figure 9 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 ms. The **best** response time for each source relation size is printed in bold font.

processors	5K	10K	50K	100K	400K
1	480	912			
3	**176**	306			
5	188	**248**	775	1416	
7	208	252	656		
10	162	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

Figure 9: Performance of selection queries

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, Figure 10 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 Figure 10. This issue is discussed after the presentation of the other performance results.

name	processors	response time	
Teradata	20	28 220	[DGS87]
GAMMA(VAX)	8	13 830	[DGS87]
Silicon DBM	3	10 900	[LeR87]
PRISMA/DB	15	735	
GAMMA(Intel)	30	450	[DGS90]

Figure 10: Response times of some parallel DBMSs to a 1% selection from 100 tuples in ms

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 LTMs. 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 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. A more formal coverage of this issue can be found in [Wil92]

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 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 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.

5.2 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 [BDT83]; 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.

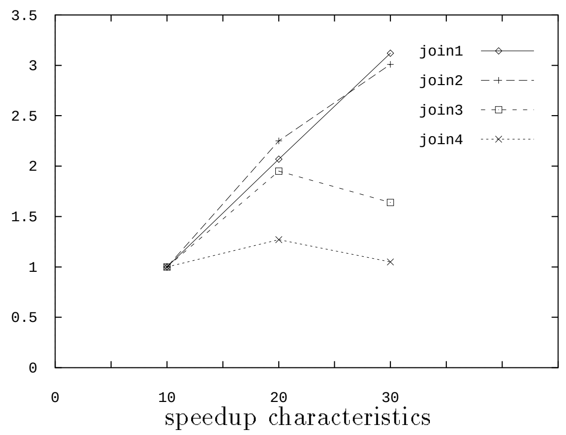
join2 The relation are fragmented in the same way as for join1, but all fragments reside on different processors. The Bprime fragments are sent to the A fragments for joining.

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 relation 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.

#	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

Figure 11: Performance of join queries

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.

Figure 11 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. Figure 12 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.

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 LTMs. 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 intra-operator parallelism does not need to be very large.

name	processors	response time	
Teradata	20	131300	[DGS87]
GAMMA(VAX)	8	45600	[DGS87]
Silicon DBM	3	23900	[LeR87]
HC16-186	16	10000	[BrG89]
GAMMA(Intel)	30	3340	[DGS90]
PRISMA/DB	30	1978	

Figure 12: Response times of some parallel DBMSs to a 100K $\times$ 10K join, fragmented on the join attribute

6 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.

6.1 Multi-join 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. Multi-join queries are used as an example of complex queries in this study. Important questions are:

- 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?

[WiA91,WiA90,WAF91] are reports on this research. In those papers, the pipelining hash-join algorithm (see Section 3.5) 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 inter-join pipelining. Its behavior in linear and bushy query plans for a restricted class of multi-join queries was studied, using simulation and analytic mathematical analysis (the distinction between left-deep and right-deep linear plans [ScD90] 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 of 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 [WiA91], secondly, we want to extend the study to a broader class of multi-join queries, and finally intra-operation parallelism for the individual join operations will be considered.

6.2 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 [SiV84], POSTRGRES [SRH90], and STARBURST [HFL89].

In this research, the basic software architecture for integrity control is based on the transaction modification principle as explained in short 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 [GrA91], 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 [GrA90]. Here, attention is paid to the translation of constraints in a functional specification (first order logic) to an operational specification in extended relational algebra (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 percents for very simple constraints and about 100 percent 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. 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 [Gre92].

7 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 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 recovery 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 multi-operation queries and parallel integrity control, as described in the previous section, will be conducted on the prototype. Further, PRISMA/DB1 is used for the implementation of parallel algorithms for transitive closure operations [HAC90,HCC91]; 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 [StA90]; 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 multi-join facilities of the system.

References

[AgD85] R. Agrawal & D. J. DeWitt, "Recovery architectures for multiprocessor database machines.," in *Proceedings of ACM-SIGMOD 1985 International Conference on Management of Data, Austin, TX, May 28–31, 1985*.

[Ame87] P. America, "POOL-T, A parallel object-oriented language," in *Oject Oriented Concurrent Programming*, A. Yonezawa & M. TOkoro,eds., MIT Press, Cambridge, MA, 1987, 199–220.

[Ame89] P. America, "Issues in the design of a parallel object-oriented language," *Formal Aspects of Computing* 1 (1989), 366–411.

[Ame91] P. America, ed., *Proceedings of the PRISMA Workshop on Parallel Database Systems*, Springer-Verlag, New York–Heidelberg–Berlin, 1991.

[AHB88] P. M. G. Apers, M. A. W. Houtsma & F. Brandse, "Processing Recursive Queries in Relational Algebra," in *Data and Knowledge (DS-2)*, R. A. Meersman & A. C. Sernadas,eds., Elsevier Science Publishers, IFIP, 1988.

[BeK91] C. A. vanden Berg & M. L. Kersten., "Engineering a Main Memory DBMS.," CWI Quarterly ?.?, Centre for Mathematics and Computer Science, Amsterdam, The Netherlands, 1991.

[BKB90] C. A. vanden Berg, M. L. Kersten & K. Blom, "A Comparison of Scanning Algorithms," in *Proceedings of the International Conference on Databases, Parallel Architectures and their Applications, Miami, USA, March 1990*.

[BCV91] B. Bergsten, M. Couprie & P. Valduriez, "Prototyping DB3S, a Shared-Memory Parallel Database System," in *Proceedings of the First International Conference on Parallel and Distributed Information Systems, Miami Beach, Florida, USA, December 1991*, 226–235.

[BDT83] D. Bitton, D. J. DeWitt & C. Turbyfill, "Benchmarking Database Systems - A Systematic Approach," in *Proceedings of Ninth International Conference on Very Large Data Bases, Florence, Italy, October 31–November 2, 1983*.

[BAC90] H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith & P. Valduriez, "Prototyping Bubba, A Highly Parallel Database System," *IEEE Transactions on Knowledge and Data Engineering* 2 (1990), 4–24.

[BoR85] H. Boral & S. Redfield, "Database Machine Morphology," in *Proceedings of Eleventh International Conference on Very Large Data Bases, Stockholm, Sweden, August 21–23, 1985*.

[BrG89] K. Bratbergsengen & T. Gjelsvik, "The Development of the CROSS8 and HC16-186 (Database) Computers.," in *Proceedings of the Sixth International Workshop on Database Machines, Deauville, France, June 1989*, 359 –372.

[BNO87] W. J. H. J. Bronnenberg, L. Nijman, E. A. M. Odijk & R. A. H. v. Twist, "DOOM: A Decentralized Object-Oriented Machine," in *IEEE Micro*.

[CeP84] S. Ceri & G. Pelagatti, *Distributed Databases, Principles and Systems*, McGraw-Hill, New York, NY, 1984.

[DGS90] D. J. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. Hsiao & R. Rasmussen, "The GAMMA Database Machine Project," *IEEE Transactions on Knowledge and Data Engineering* 2 (March 1990), 44–62.

[DGS87] D. J. DeWitt, S. Ghandeharizadeh, D. Schneider, R. Jauhari, M. Muralikrishna & A. Sharma, "A single user evaluation of the GAMMA Database Machine," in *Proceedings of the Fifth International Workshop on Database Machines, Karuizawa, Japan, October 1987*.

[DKO84] D. J. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebreaker & D.Wood, "Implementation techniques for main memory database systems.," in *Proceedings of ACM-SIGMOD 1984 International Conference on Management of Data, Boston, MA, June 18–21, 1984*, 1–8.

[Eic87] M. Eich, "A classification and comparison of main memory database recovery techniques," in *Proc. of the 1987 Database Enginering Conference, 1987*, 332–339.

[GLH83] H. Garcia-Molina, R. J. Lipton & P. Honeyman, "A Massive Memory Database System," Technical Report 314, Department of Comp Science, Princeton University, September 1983.

[GaV89] G. Gardarin & P. Valduriez, *Relational Databases and Knowledge Bases*, Addison-Wesley, Reading, MA, 1989.

[Gre92] P. W. P. J. Grefen, "Dynamic Action Scheduling in a Parallel Database System," in *Proceedings of the Conference on Parallel Architectures and Languages in Europe, Paris, France, 1992*.

[GrA90] P. W. P. J. Grefen & P. M. G. Apers, "Parallel Handling of Integrity Constraints on Fragmented Relations," in *Proceedings of the Second International Symposium on Databases in Parallel and Distributed Systems, Dublin, Ireland, July 2-4 1990*, 138 – 145.

[GrA91] P. W. P. J. Grefen & P. M. G. Apers, "Integrity Constraint Enforcement through Transaction Modification," in *Proceedings 2nd International Conference on Database and Expert Systems Applications, Berlin, Germany, July 1991*.

[GWF91] P. W. P. J. Grefen, A. N. Wilschut & J. Flokstra, "PRISMA/DB1 User Manual," Memorandum INF91-06, Universiteit Twente, Enschede, The Netherlands, 1991.

[HFL89] L. Haas, J. C. Freytag, G. Lohman & H. Pirahesh, "Extensible Query Processing in Starburst," in *Proceedings of ACM-SIGMOD 1989 International Conference on Management of Data, Portland, OR, May 31–June 2, 1989*.

[HAC90] M. A. W. Houtsma, P. M. G. Apers & S. Ceri, "Distributed Transitive Closure Computations: The Disconnection Set Approach.," in *Proceedings of Sixteenth International Conference on Very Large Data Bases, Brisbane, Australia, August 13–16, 1990*, 335–346.

[HCC91] M. A. W. Houtsma, F. Cacace & S. Ceri, "Parallel Hierarchical Evaluation of Transitive Closure Queries," in *Proceedings of the First International Conference on Parallel and Distributed Information Systems, Miami Beach, Florida, USA, December 1991*.

[Ker89] M. L. Kersten, "Using logarithmic code-expansion to speedup index access.," in *Foundations of Data Organization and Algorithms, INRIA, Springer-Verlag, June 1989.*, 228–232.

[Kui91] E. van Kuijk, *Semantic Query Optimization in Distributed Database Systems*, PhD-Thesis, University of Twente, 1991.

[LeC86] T. J. Lehman & M. J. Carey, "Query processing in main memory database management systems.," in *Proceedings of ACM-SIGMOD 1986 International Conference on Management of Data, Washington, DC, May 28–30, 1986*, 239–250.

[LeC87] T. J. Lehman & M. J. Carey, "A recovery algorithm for a high-performance memory-resident database system.," in *Proceedings of ACM-SIGMOD 1987 International Conference on Management of Data, San Francisco, CA, May 27–29, 1987*.

[LeR87] M. D. P. Leland & W. D. Roome, "The Silicon Database Machine: Rational, Design, and Results," in *Proceedings of the Fifth International Workshop on Database Machines, Karuizawa, Japan, October 1987*.

[ScD90] D. A. Schneider & D. J. DeWitt, "Tradeoffs in Processing Complex Join Queries via Hashing in Multiprocessor Database Machines," in *Proceedings of Sixteenth International Conference on Very Large Data Bases, Brisbane, Australia, August 13–16, 1990*, 469–480.

[SiV84] E. Simon & P. Valduriez, “Design and Implementation of an Extendible Integrity Subsystem,” in *Proceedings of ACM-SIGMOD 1984 International Conference on Management of Data, Boston, MA, June 18–21, 1984*.

[Spe91] J. vander Spek, “POOL-X and its implementation,” in *Proceedings of the PRISMA Workshop on Parallel Database Systems, Noordwijk, The Netherlands, 1990*, P. America,ed., Springer-Verlag, New York–Heidelberg–Berlin, 1991, 309–344.

[StA90] H. J. Steenhagen & P. M. G. Apers, “ADL - An Algebraic Database Language.,” in *Proceedings Computing Science in the Netherlands, Utrecht, the Netherlands, november 1990*, 427 – 442.

[Sto75] M. Stonebraker, “Implementation of Integrity Constraints and Views by Query Modification,” in *Proceedings of the ACM-SIGMOD 1975 International Conference on Management of Data, San Jose, USA, 1975.*.

[SKP88] M. Stonebraker, R. Katz, D. Patterson & J. Ousterhout, “The Design of XPRS,” in *Proceedings of Fourteenth International Conference on Very Large Data Bases, Los Angeles, CA, August 29–September 1, 1988*.

[SRH90] M. Stonebraker, L. A. Rowe & M. Hirohama, “The Implementation of POSTGRES,” *IEEE Transactions on Knowledge and Data Engineering* 2 (March 1990).

[WaT91] P. Watson & P. Townsend, “The EDS Parallel Relational Database System,” in *Proceedings of the PRISMA Workshop on Parallel Database Systems, Noordwijk, The Netherlands, 1990*, P. America,ed., Springer-Verlag, New York–Heidelberg–Berlin, 1991.

[Wil92] A. N. Wilschut, “Parallelism and Performance in PRISMA/DB,” Memorandum INF92-19, Universiteit Twente, Enschede, The Netherlands, 1992, Submitted for publication.

[WiA91] A. N. Wilschut & P. M. G. Apers, “Dataflow Query Execution in a Parallel Main-Memory Environment,” in *Proceedings of the First International Conference on Parallel and Distributed Information Systems, Miami Beach, Florida, USA, December 1991*.

[WiA90] A. N. Wilschut & P. M. G. Apers, “Pipelining in Query Execution,” in *Proceedings of the International Conference on Databases, Parallel Architectures and their Applications, Miami, USA, March 1990*.

[WAF91] A. N. Wilschut, P. M. G. Apers & J. Flokstra, “Parallel Query Execution in PRISMA/DB,” in *Proceedings of the PRISMA Workshop on Parallel Database Systems, Noordwijk, The Netherlands, September 1990*, P. America,ed., Springer-Verlag, New York–Heidelberg–Berlin, 1991.

[WGA89] A. N. Wilschut, P. W. P. J. Grefen, P. M. G. Apers & M. L. Kersten, “Implementing PRISMA/DB in an OOPL.,” in *Proceedings of the Sixth International Workshop on Database Machines, Deauville, France, June 1989*, 359 –372.