KARDAMOM — A Dataflow Database Machine For Real-Time Applications

Günter von Bültzingsloewen, Klaus R. Dittrich, Cirano Iochpe,
Rolf-Peter Liedtke, Peter C. Lockemann, Michael Schryro

Forschungszentrum Informatik
an der Universität Karlsruhe
Haid-und-Neu-Straße 10–14
D–7500 Karlsruhe 1
email: liedtke@germany.csnet

Abstract: *In real-time applications, database systems have to cope with requirements different from those prevailing in a commercial environment. High performance in the presence of complex queries and high reliability are the most crucial issues. To comply with these, it seems promising to adopt techniques which have been proposed in the context of database machines, such as parallel query processing, filtering, and data flow control. In this paper we present the concepts and the design of the data flow database machine KARDAMOM which is mainly geared towards real-time applications and is currently being developed at FZI Karlsruhe.*

1 Introduction

In real-time applications, database systems have to cope with requirements which are rather different from those prevailing in a commercial environment [Lock85]. But as "the" real-time application which could be used as a reference does not exist, it is hardly possible to determine a set of typical requirements. Characteristic properties like database size, query rate, response time etc., vary considerably. However, there are some issues which seem to be common to a majority of real-time applications, including

- predictable response time (by definition the key property of real-time systems)
- high performance (i.e. short response times, high throughput, spare computing power for peak query loads) in the presence of transactions that are more complex than simple debit-credit transactions
- high reliability
- scalability (e.g. adaptability of performance and functionality) with regard to the demands of an individual application

As performance and reliability are the most crucial issues, a database backend approach separating the database system from the rest of the application seems to be most appropriate. Thus the application system is offloaded while the database system can be designed and optimized independently. Especially, exploiting parallelism in the database backend is a promising method to improve both, performance and reliability.

Within the KARDAMOM project at FZI Karlsruhe we are currently developing a prototype of such a parallel backend database system for real-time applications. Our approach is based on recent research in database machines and integrates several concepts from this area. In the remainder of this paper, we give an overview of these concepts and of the architecture of the KARDAMOM database machine.

2 Concepts And Architectural Overview

Many database machine approaches such as RDBM [Schw83], VERSO [Banc83] or GRACE [FKTa86] are based on specially developed hardware devices (filter processors, hardware sorter). However, this has proved to be expensive and difficult to develop [Schw83]. Other approaches choose standard hardware and develop new concepts for parallelizing and controlling the database software, such as MDBS [HeHi83] and GAMMA [DeWi86]. These include data partitioning with parallel disk access for increased I/O-bandwidth, distributed query evaluation with intra-query parallelism, in particular parallel join algorithms, and data flow control for high parallelism and reduced control overhead. For reasons of economy, we follow the latter approach and have chosen a multi-microprocessor system built from off-the-shelf components as the hardware basis for the database machine. Thus the design concentrates upon the appropriate database software.

Our database machine provides a relational interface with a slightly extended SQL as the query language. The amount of host/backend communication is kept low as SQL statements are precompiled and serve as the units of communication. This is important as communication might become a bottleneck if low level data access statements were used instead [DrSc83].

The basic idea behind the software design is to decompose the database management system into a number of functional components, each implementing specific suboperations of transaction processing (relational algebra operations, software filtering, update, synchronization and recovery). The components are realized as processes that are mapped onto the processors of the multiprocessor system. The assignment of components to processors is handled in a very flexible way. It is possible to have only a single instance of some component mapped to some processor as well as to have several instances of other components which are replicated on several processors. Copies of the same component can process different operations (e.g. several centralized join operations originating from different database operations) or they can cooperate in processing a single operation (e.g. a distributed join operation originating from a single database operation). Thus we are able to obtain intra-transaction parallelism for reduced transaction response times as well as inter-transaction parallelism for increased throughput. Furthermore, the components can be replicated and distributed across the processors as demanded by the performance requirements of a special application. The operation of the functional components is controlled by data flow in order to achieve a high degree of parallelism as well as a minimal control overhead.

For secondary storage access software filtering is used. Relations are scanned sequentially, and primitive relational operations such as simple selections and projections are applied to the tuples while they are read from disk (thus overlapping I/O operations with the first step of query evaluation). The result of a filter operation consists of those tuples satisfying the given selection expression; only these are transferred to main memory.

In contrast to conventional page-oriented database buffers where every page containing at least one qualified tuple is loaded, it is not appropriate in our case to maintain the page organization of the secondary storage in main memory. Instead, we have to deal with results of filter operations which are represented by subsets of base relations. Therefore, we introduce a set-oriented memory management mechanism. More efficient utilization of main memory is achieved by retaining only those sets of tuples actually needed by the functional components. Furthermore, communication overhead is reduced because again only these tuples are transmitted between functional components.

According to these concepts, we can distinguish two kinds of data objects. First, we have so-called *physical objects*, representing the base relations defined in the schema. Second, we have tuple sets which are the results of suboperations performed by the functional components. These are called *logical objects*. Accordingly, we can build two groups of functional components and thus decompose the database management system into two subsystems that communicate via the set-oriented memory manager. While the *logical system* DBMS_{log} contains all functional components operating on logical objects, the *physical system* DBMS_{phys} realizes all functions that map logical objects to physical objects and vice versa (see Figure 1).

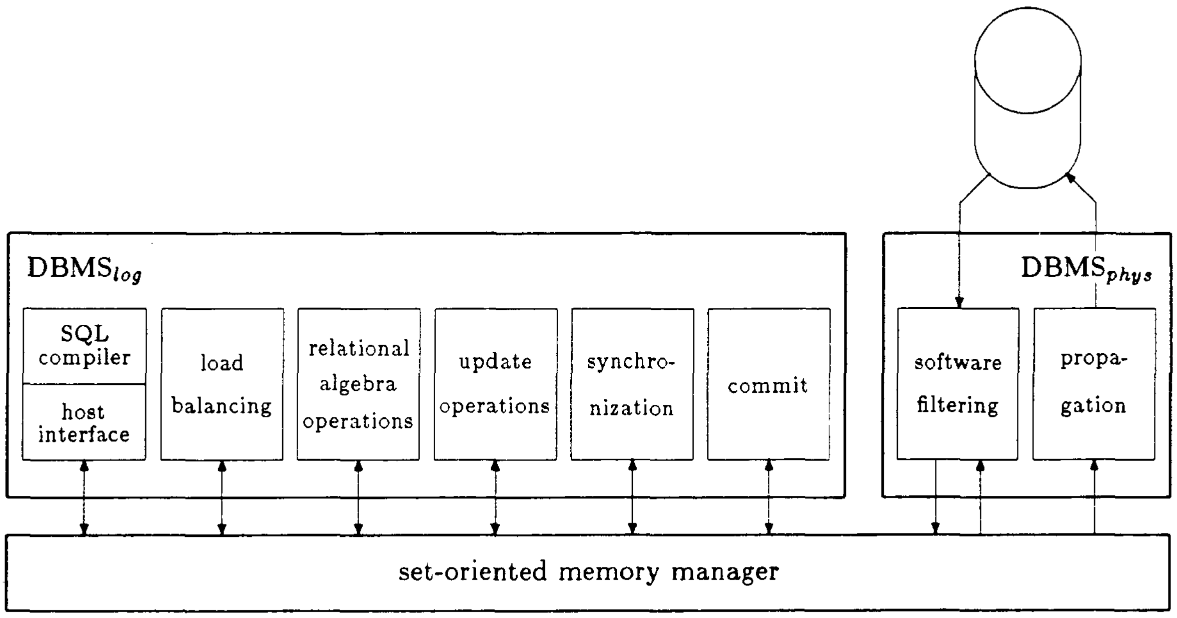

Figure 1: Architecture of the database management system

In the subsequent sections we will present the components and mechanisms of the system in some more detail.

3 Precompilation of SQL Queries

An SQL query is precompiled into a data flow program which corresponds to the operator tree of a relational algebra query [BoDe82] and contains exactly the operations that can be executed by the functional components. In order to be as expressive as SQL and to support optimization, the underlying relational algebra has to be extended (see e.g. [Bült87, Daya87]).

The data flow program can first be optimized using mostly traditional techniques. Afterwards it can be modified by node splitting and pipelining to achieve an optimal degree of parallelism, thus guaranteeing the required response time provided the machine is not overloaded.

4 The Physical System

The physical system creates logical objects from base relations and updates base relations.

As base relations are primarily held on secondary memory, the DBMS_{phys} has to manage and access secondary storage. As mentioned above, read access, i.e. the creation of logical objects, is accomplished by software filtering. As filtering basically means searching a whole relation it may be time-consuming even if all pages of the relation are stored contiguously. Therefore, we provide some techniques to accelerate filtering. First, in addition to the so-called exhaustive filtering, index-supported filtering for queries with high selectivity is available (in analogy to [Kies83]). Second, besides the parallelism which is inherent to the filter technique, more parallelism is introduced by distributing the database across several disks and providing a separate filter processor for each disk. Then disjoint partitions of the same relation can be searched in parallel. As a special case, it is possible to establish mirror disks. This adds to reliability by redundant storage as well as to improved performance by parallel filtering.

Write access integrates logical objects created by insert or update operations into the corresponding base relations (i.e. propagation [HaRe83]). As secondary storage access is page-oriented, the DBMS_{phys} has to perform a mapping from the set-oriented to the page-oriented representation of data objects.

Transactions exist in the DBMS_{log} only. Operations of the DBMS_{phys} are invoked by the DBMS_{log}, but are not themselves executed under control of the transaction management. Particularly, writing updated pages to disk is performed completely independently of the transaction causing the update. Hence, the DBMS_{phys} is an autonomous component and thus is able to implement optimizing access strategies supporting an efficient access of base relations. Some more details are given in section 8.

5 The Logical System

The DBMS_{log} contains all functional components operating on logical objects, i.e. relational algebra operations on tuple sets created by the DBMS_{phys} and update operations including insertion and deletion. The former include operations for the realization of node splitting, for example the partitioning necessary for parallel join algorithms [DeGe85]. The latter create tuple sets, which are integrated into base relations by the DBMS_{phys} after the corresponding transaction has been committed. Furthermore, according to the design of the DBMS_{phys}, the whole transaction management is included in the DBMS_{log}.

Another important task of the DBMS_{log} is load balancing. It distributes the operations of a data flow program derived from an SQL query across processors running the appropriate functional components. After distribution, the operations are executed under data flow control.

6 Data Flow Control

The load balancing component distributes operations by sending them to operation schedulers (one on each processor) that implement data flow control. Their task is to realize the firing rule for the operations they have received. It is fulfilled as soon as all operands needed for the execution of the operation are available. If the firing rule for an operation is fulfilled and a functional component implementing the operation is available, it can be sent to this component for execution. In case the firing rule is fulfilled for several operations of one or more dataflow programs, the scheduler has to select one.

After an operation has been executed, the operation schedulers supervising succeeding nodes are informed that the corresponding operands are available. The operands themselves are directly transmitted between the functional components by means of the set-oriented memory manager.

7 The Set-oriented Memory Manager

As secondary memory access is based on filtering, retrieval operations have the effect that only subrelations are transferred to main memory. The same is true for update operations which select tuples qualifying for update or deletion. Similarly, tuples to be inserted into the database can be represented as sets. All functions of the DBMS_{log} thus operate on tuple sets, i.e. operands and results of operations of the functional components are sets in general. Therefore, an appropriate representation of such set objects is needed. The set-oriented memory manager provides a uniform functionality not only to manage and access set objects but also to communicate them between functional components. As set objects are referenced by logical names, communication can remain completely transparent for the individual functional components. The set-oriented memory manager autonomously decides about the need of transferring set objects and executes the transfers if necessary. More details on the set-oriented memory management can be found in [BLDi87].

8 Transaction Management

As the architecture of our database management system consists of two levels, the DBMS_{log} and the DBMS_{phys}, we can use a two-level concurrency control mechanism [Weik86]: the DBMS_{log} has to guarantee that transactions are serializable with respect to calls of operations of the DBMS_{phys}; furthermore, the operations of the DBMS_{phys} must be atomic, i.e. serializable in terms of page accesses.

Read and write operations of the DBMS_{phys} (software filtering and integration of logical objects created by update operations into base relations, respectively) can be synchronized using precision locks [JBBa81] in combination with the (r, a, x)-protocol.

We also use a two-level logging and recovery mechanism. To commit a transaction, all its updated logical objects are saved as redo-information on the logical log (LLog) that is controlled by the DBMS_{log}. The corresponding x-locks are maintained until the (logical) integration of the updates into base relations is completed. After mapping the updates of a write operation to pages, the DBMS_{phys} saves its updated mapping information (tuple tables, page tables, indices) on the physical log (PLog). When eventually all primary data pages containing updated tuples have been propagated to disk, the logical system is informed that the corresponding LLog entry can be deleted.

For recovery after a system crash, the physical system reads PLog and installs the mapping information. In parallel, the logical system reads LLog backwards and creates logical objects which redo the updates possibly lost due to the crash. The following write operation of the physical system has to be idempotent, as some updated tuples may have been propagated to disk already.

There are several advantages to our approach to transaction management: the two-level concurrency control reduces conflict probability in comparison with a traditional one-level mechanism, a transaction is committed as early as possible, the physical system can decide independently when an update is mapped to pages and when modified pages are propagated to disk, and the amount of logged data is kept at a minimum.

9 Future Work

A one-processor version of the proposed architecture is currently (end of 1987) close to completion within the KARDAMOM project at FZI Karlsruhe. Our future work will focus on problems arising in a multiprocessor architecture. The issues we want to explore in some detail include the following:

- The query optimizer can modify a data flow program at compile time by node splitting and pipelining. As increased parallelism within a data flow program also means increased control overhead at execution time, an optimal degree of parallelism has to be found: it should guarantee the required response time and at the same time minimize the overall use of system resources. We will investigate how this optimization objective can be formalized and which search strategies and algorithms can be used within the optimizer to meet it.
- Data flow control should schedule operations such that the required response times of data flow programs are met. The load balancing component and the operation schedulers have to use appropriate strategies; these might use scheduling information (e.g. operation deadlines, priority lists) determined by the optimizer.
- Set-oriented memory management poses several further problems. These include communication between functional components which is needed to transmit set objects according to data flow. Another topic is a replacement strategy for the case that set objects cannot be kept in main memory because of space limitations.
- Our approach to transaction management allows to defer updates arbitrarily. Hence the choice of an appropriate point in time — with respect to synchronization of concurrent transactions — to map a set-oriented representation of an update result to the respective database pages is another important topic. It affects throughput as well as transaction response time.
- The database may be distributed across several disks to support parallel filtering. Here the problem arises to find out which distribution strategy is best, i.e. how relations should be partitioned and distributed to achieve a maximum of efficiency.

Bibliography

[Banc83] F. Bancilhon et al.: VERSO: A Relational Backend Database Machine. In: D.K. Hsiao (ed.): Advanced Database Machine Architecture. Prentice-Hall, 1983

[BLDi87] G. v. Bültzingsloewen, R.-P. Liedtke, K. R. Dittrich: Set-Oriented Memory Management In A Multiprocessor Database Machine. Proc. 5th Int. Workshop on Database Machines, Karuizawa, October 1987, pp. 611–625

[BoDe82] H. Boral, D.J. DeWitt: Applying Data Flow Techniques to Database Machines. IEEE Computer, August 1982, pp. 57–63

[Bült87] G. v. Bültzingsloewen: Translating and Optimizing SQL Queries Having Aggregates. Proc. 13th Int. Conf. on Very Large Data Bases, Brighton, September 1987

[Daya87] U. Dayal: Of Nests and Trees: A Unified Approach to Processing Queries That Contain Nested Subqueries, Aggregates and Quantifiers. Proc. 13th Int. Conf. on Very Large Data Bases, Brighton, September 1987

[DeGe85] D.J. DeWitt, R. Gerber: Multiprocessor Hash-Based Join Algorithms. Proc. 11th Int. Conf. on Very Large Data Bases, Stockholm, 1985

[DeWi86] D.J. DeWitt et al.: GAMMA - A High Performance Dataflow Database Machine. Proc. 12th Int. Conf. on Very Large Data Bases, Kyoto, August 1986

[DrSc83] M. Drawin, H. Schweppe: A Performance Study on Host-Backend Communication. Proc. 3rd Int. Workshop on Database Machines, Munich, September 1983

[FKTa86] S. Fushimi, M. Kitsuregawa, H. Tanaka: An Overview of The System Software of A Parallel Relational Database Machine GRACE. Proc. 12th Int. Conf. on Very Large Data Bases, Kyoto, August 1986

[Gard83] G. Gardarin et al.: SABRE: A Relational Database System for a Multimicroprocessor Machine. In: D.K. Hsiao (ed.): Advanced Database Machine Architecture. Prentice-Hall, 1983

[HaRe83] T. Härder, A. Reuter: Principles of Transaction-Oriented Database Recovery. ACM Computing Surveys, Vol. 15, No. 4, December 1983, pp. 287–317

[HeHi83] X. He, M. Higashida et al.: The Implementation of a Multibackend Database System (MDBS): The Design of a Prototype MDBS. In: D.K. Hsiao (ed.): Advanced Database Machine Architecture. Prentice-Hall, 1983

[JBBa81] J.R. Jordan, J. Banerjee, R.B. Batman: Precision Locks. Proc. ACM-SIGMOD Int. Conf. on Management of Data, 1981

[Kies83] W. Kiessling: Database Systems for Computers with Intelligent Subsystems: Architecture, Algorithms, Optimization. Report TUM-I8307, Technical University of Munich, Institute of Computer Science, August 1983 (in German)

[Lock85] P. C. Lockemann et al.: Database Requirements of Engineering Applications — An Analysis. FZI publication No. 3, Forschungszentrum Informatik an der Universität Karlsruhe, July 1985

[Schw83] H. Schweppe et al.: RDBM - A Dedicated Multiprocessor System for Database Management. In: D.K. Hsiao (ed.): Advanced Database Machine Architecture, Prentice-Hall, 1983

[Weik86] G. Weikum: A Theoretical Foundation of Multi-Level Concurrency Control. Proc. 5th Symp. on PODS, Cambridge(Mass.), 1986