DataCube: An Integrated Data and Compute Server Based On A Hypercube-Connected Dataflow Database Machine

Goetz Graefe

Oregon Graduate Center
Department of Computer Science
and Engineering
19600 N.W. von Neumann Drive
Beaverton, OR 97006-1999 USA

Technical Report No. CS/E 88-024

Abstract

Parallel hardware has been used for both compute-intensive applications and data management, typically as a server for a number of workstations. If both services were needed, only one of them was parallelized, or two servers were connected via a network. Instead, we propose to use one highly parallel hardware base and to build system software that provides three abstractions; a compute server, a database machine, and an integrated data and compute server. The main goal of the integrated server architecture is to exploit parallel communication between servers, and to study the resource management problems of providing both services by all available processors. The system can also be used to introduce transactions and checkpoints to compute-intensive applications. Since hypercubes are suitable for both kinds of servers, we will by using hypercube-connected hardware, and call our project DataCube.

DataCube:
An Integrated Data and Compute Server
Based On
A Hypercube-Connected Dataflow Database Machine

Motivation and Preliminary Design

Goetz Graefe
Oregon Graduate Center
graefe@cse.ogc.edu

Abstract

Parallel hardware has been used for both compute-intensive applications and data management, typically as a server for a number of workstations. If both services were needed, only one of them was parallelized, or two servers were connected via a network. Instead, we propose to use one highly parallel hardware base and to build system software that provides three abstractions: a compute server, a database machine, and an integrated data and compute server. The main goal of the integrated server architecture is to exploit parallel communication between servers, and to study the resource management problems of providing both services by all available processors. The system can also be used to introduce transactions and checkpoints to compute-intensive applications. Since hypercubes are suitable for both kinds of servers, we will be using hypercube-connected hardware, and call our project DataCube.

1. Introduction

Many applications require extensive computations on large datasets, e.g., weather forecasting, climatology, speech and image processing. In order to improve the response time in these applications, many attempts have been made to exploit parallelism. In almost all cases, however, the parallelism was used either in computation or in data management, but not in both. For example, most of the large compute engines have only rudimentary data management facilities, and database machines have only very limited computational power, typically some minor extensions to the relational algebra. If in fact both services, extensive computations and data management, are parallelized, they are parallelized separately, e.g., a powerful compute engine is connected via a "thin wire" such as an ethernet with a database machine. In other cases, a parallel hardware base is used for both, but only for one at any given time. In applications requiring both services, we believe that tighter integration is the key to high performance,

low response time, high throughput, and high hardware utilization. In other words, instead of connecting a highly parallel database machine and a powerful compute server with one thin wire, we propose to combine both services in one system architectures. The obvious advantages are twofold. First, the typical case that large computations operate on large datasets can be supported more efficiently because there are many, short datapaths between the disk storage and the compute engines instead of a single long datapath. Second, load can be spread across more computers. Thus, while no compute intensive application is running, the database machine can make full use of all CPU's in the system, and while no database application is running, the compute intensive application can be distributed across twice as many CPU's as if two separate servers were used.

A number of hardware architectures have been proposed both for computation-intensive applications and for data management. In a highly parallel database machine, low control overhead and even distribution of dynamic load seem to be crucial to high performance [DeWitt1986a, Alexander1988a]. Thus, a simple but fast interconnection of processing elements is crucial in such a database machine. Hypercube architectures provide a very regular, yet flexible and powerful interconnection network between homogeneous processors. While this architecture has been employed very successfully as "number crunching" vehicle for such diverse applications as computational mechanics and neural network simulation, database algorithms have not been sufficiently explored on hypercubes. One of the reasons is that it has been shown only recently that high degrees of parallelism can be made to work for database queries. Another reason is that, until recently, hypercubes provided ample processing power, but not the I/O transfer rate to keep the processors utilized. Through the addition of local I/O facilities and file system software to individual nodes in hypercube multi-computers, this architecture becomes a very attractive vehicle for database applications. Since hypercubes have been used so far typically as compute server for a set of workstations, it is the logical next step to develop a hypercube-based integrated data and compute server.

In this paper, we present the initial design for an integrated data and compute server. The goal is that the system software running on a parallel hardware base provides three abstractions. Users on workstations can view the integrated data and compute server as pure compute server, i.e., the traditional view of a supercomputer, as pure data server, i.e., a database machine, or as an integrated data and compute server. Our main interest is exploiting the parallel communication capabilities of the hypercube multi-processor architecture for data transfer within and between the servers, and in studying the resource management problems in an integrated server.

The software will employ a hypercube of any size, typically 2^3 to 2^{10} processors. A number of these processors have local disk storage attached. The number of disks in the entire system does not need to be a power of 2. We plan on attaching a disk to about half the processors as the typical case. For our work, we assume a hypercube with an operating system kernel that provides multi-threading (processes), reliable communication, and local disk I/O. Other operating system services, like the process structure, memory management, and the file system, have been or will be custom built. Our effort is software-oriented rather than hardware-oriented. In this respect, it is similar to another recently developed database machine, GAMMA [DeWitt1986a]. A number of differences between DataCube and GAMMA are a result of the different processor interconnection (a cube rather than a token ring) which allows defining and exploiting distance between processors. Other differences are in our strategies for partitioning intermediate result relations, using local strategies in some cases, and, as pointed out above, the fact that the DataCube database machine will become part of a larger, integrated server environment.

The remainder of this paper is organized as follows. In Section 2, we briefly discuss other database machine efforts, and point out directions found in more recent designs. In Section 3, we outline the problems of integrating data and compute services on one parallel hardware base, and describe the approaches we will use to solve them. Section 4 gives a detailed account

of our design principles for database processing algorithms which are applied in Section 5 to develop algorithms for relational join and division, for aggregate functions, for duplicate elimination, and for parallel sorting. Section 6 contains some considerations for query optimization, emphasizing short communication paths to allow parallel communication in the cube-connected database machine. In Section 7, we list very briefly some of our ideas how we will provide transactions in the integrated server environment. Section 8 contains a summary and our conclusions.

2. Database Machines

With a clear understanding of what functionality a database management system should provide, e.g., interpretation of relational algebra or calculus [Codd1970a, Codd1972a], came the idea to off-load mainframe computers by using a dedicated machine for database management. In the 1970's, most proposed and implemented designs included one or more pieces of special design hardware, e.g., hardware sorters or intelligent disks [Su1975a, Ozkarahan1977a]. Furthermore, a number of researchers experimented with parallel hardware, e.g., [DeWitt1979a]. However, an influential paper by Boral and DeWitt [Boral1983a] identified a number of problems with these approaches. First, as the performance of general purpose CPU's is improved at a much faster rate than the transfer rate of disk devices is improved, parallel processing can only pay off if high data rates for retrieval and selection can be achieved. Thus, the performance of highly parallel database machines depends on the bandwidth between permanent storage devices and processing elements and among processing elements. Second, designing and implementing specialized hardware delay project development and adaptation to technological advances such that software ported to new general purpose CPU's may outperform the latest available version of specialized hardware.

As a consequence, more recent database machine designs rely on few if any special purpose hardware components [Teradata1983a, DeWitt1986a, Alexander1988a]. Since we are most familiar with and influenced by the GAMMA project, we discuss it here in some detail. GAMMA is

a software architecture running on a number of general purpose computers as a backend to a UNIX host machine. Currently, it runs on 17 VAX 11/750's connected with each other and the VAX 11/750 host via a 80 Mb/s token ring. Eight GAMMA processors have a local disk device. The disks are accessible only locally, and update and selection operators use only these 8 processors. The other, diskless processors are used for join processing. GAMMA extensively uses hash-based algorithms, implemented in such a way that each operator is executed on several (or all) processors and the input stream for each operator is partitioned into disjoint sets according to a hash function [DeWitt1985a, DeWitt1986a, Gerber1986a]. A detailed description and a preliminary performance evaluation of GAMMA can be found in [DeWitt1986a], more information on its performance in [DeWitt1988a].

Baru and Frieder introduced dynamic data re-distribution in their hypercube database machine design [Baru1987a]. We believe, however, that re-distribution will decrease rather than increase system performance. It seems more important to design reliable optimization and estimation techniques to avoid the need for dynamic re-distribution. Furthermore, their model does not include partitioning, hash-based algorithms, or indices, all of which DataCube supports.

Bratbergsengen is currently porting single operator algorithms [Bratbergsengen1984a] from the Cross-8 database machine to hypercube connected hardware, with justified expectations of excellent performance. While the individual algorithms are basically the same, the DataCube effort goes far beyond this work as it integrates a database machine supporting complex queries, concurrent queries, and query optimization in a dual-server architecture.

3. Integration of Data and Compute Servers

Before we consider architectural questions for an integrated data and compute server, we will briefly describe how it can be useful. We see basically five reasons. First and foremost, we believe that these services are used typically together. In all applications using a large dataset, a fair amount of processing is required to *extract* or *infer* from the stored data the information required to satisfy user demands, and to *translate* or *present* the information contained in the

data suitably to the user. The other way around, applications requiring large computations also require large amounts of data, e.g., weather forecasting requires current weather data, geophysical data, and historical data of past weather patterns and developments [NCAR1987a]. Second, short datapaths with high bandwidth between permanent storage and processing locations are important for most applications, in particular for real-time applications. Third, economic reasons are a strong incentive for providing both services using a shared hardware base, each with equal or increased power for the same total investment. Fourth, data servers such as database machines need a fair amount of processing power to maintain the database, not only for query processing but also for consistency and integrity enforcement[1] and for maintenance of storage structures. We expect that new applications such as design databases with complex objects will allow and enforce more sophisticated and computationally expensive integrity constraints. Fifth, compute servers and supercomputers need I/O and data services for working storage and for logging and checkpointing, i.e., for keeping an update and activity trace and saving a fallback state on stable storage for the case of subsequent failure.

We see a number of challenges in integrating data and compute services on a single platform of parallel hardware. They can be broadly classified into outside control, synchronization, communication and data formats, resource management, and failure detection and recovery. We are reviewing these problem areas and our approaches to them in turn. While appropriate command languages and subsystem interfaces have been designed and implemented before, for either data servers or computer servers, the control language for an integrated server also needs to allow control over load distribution, synchronization of dependent activities, and data interchange between the two server subsystems independently of (i.e., bypassing) the user's workstation.

---

[1] Some promising new database system designs *assume* ample and sometimes idle processing power, e.g., for the *vacuum daemon* in POSTGRES [Stonebraker1987a].

If the data is partitioned on some field value and the query includes a restriction on this field, the query pertains only to a small number of data nodes (i.e., nodes with attached disk drives), and thus induces a load imbalance in the system. For a complex query with multiple repartitioning steps, e.g., multiple joins [DeWitt1986a], the database subsystem may be able to ensure load balance. In some cases, however, it will be desirable that scheduling within the compute-server offsets the load imbalance of the database activities. In other cases, database scheduling may need to counterbalance unevenly distributed compute-server load. We will explore possibilities to use a single scheduler which coordinates both servers' activities. This could be a database scheduler augmented to accept "foreign" processes, i.e., processes using compute-services of the integrated server. Necessary modifications to the scheduler include developing and implementing a suitable protocol for registering foreign processes and their anticipated resource needs with the scheduler.

Assuming that the load distribution problems can be addressed satisfactorily, consider the synchronization of interdependent activities of the two subsystems, in particular data exchange between them. First, the distribution of data must be determined, i.e., which data items are exchanged at which node and between which processes. Recall that distributed, parallel data exchange between data and compute servers is a primary motivation for their integration. Second, the exchange paradigm must be agreed upon: the main distinction here is demand- or data-driven dataflow. Third, while most database algorithms are tuned to maximize throughput and minimize response time, a compute intensive application may not be able to use the data as fast as the database subsystem can deliver them, and thus the compute server needs to apply *back-pressure* beyond the data exchange point to a data-driven dataflow database subsystem. At this point in time, we do not know appropriate answers to these problems. Our initial approach will be to use demand-driven dataflow locally, both within the database machine and between the servers, and use data-driven dataflow for data partitioning within the database subsystem, augmented with a distributed back-pressure mechanism.

The data exchange format and granularity must be controllable by the user. We intend to augment the database query language with format specification, maybe similar to specifications used in current database systems for dumping to and loading from external files. The granularity of data exchange can be a record, a block, the entire dataset, etc. We expect that the granularity will have a significant impact on performance and resource consumption of communication-intensive database queries and computations, and intend to investigate its influence carefully.

Resource management has been touched upon above in connection with load balancing. Other issues that must be addressed in the design of an integrated data and compute server are global fair scheduling, task and user priorities, memory allocation (I/O buffers, database work space, e.g., hash tables, and program stack and heap space), protection and security, isolation of failures, adaptation to configuration changes and local failures, and more. Resource management will be an important part of our research, both policies to arbitrate resource contention, and mechanisms to implement and evaluate possible policies.

4. General Outline of the Software Architecture

In the remainder of this report, we focus on the design of the data server subsystem, which is essentially a high-performance database machine. In order to make best use of the available processing power in a hypercube, more than one query can be active in DataCube at any given time, more than one operator can be active within a single query, and more than one machine can work on a single resource-intensive operator, e.g., an aggregate function or a join. The input data for each complex operator are divided into disjunct subsets called partitions, one of which is assigned to each machine. Between two operations, e.g., between two joins, the data can be repartitioned across the interconnection network. It is important to note that the set of machines working on the first operator and the set of machines working on the second operator may intersect or even be equal, e.g., all machines in the cube. In the latter case, data exchange between machines may still be necessary if the second operator requires that the data be parti-

tioned on a different attribute.

If two subsequent operators within a single query are executed in parallel, the first operator, called the producer in this relationship, sends a page of data as soon as a page frame is filled with output data. The second operator, called the consumer, reads data off the network, processes them, and sends results to the next operator in the query as quickly as possible. Since intermediate results are never collected within one operator (if it is avoidable), we call DataCube a *dataflow database machine*. Using the dataflow architecture, it is possible that the first data reach the host or user terminal before all file scans are completed.

It should be apparent that sorting does not match well with these processing strategies. Instead, we will rely on hashing whenever possible. Sorting is implemented for three reasons, however. First, it may be explicitly requested for presentation to the user. Second, sorting is necessary for repartitioning on key ranges and for clustering. Third, we are not aware of parallel sorting methods for very large datasets, and intend to develop new methods in DataCube.

When evaluation of a query is initiated, query packets are sent to the processing nodes by a query scheduler process that acts as a coordinator for one query. Several such query scheduler processes may be active in DataCube. Each packet represents a small subtree of operators participating in the query evaluation. Upon receipt of a query packet, a node performs the requested processing steps and then reports back to the query scheduler. A leaf of a local subtree is either a scan (file or index scan) on a local disk or a *receive* operator. The root of a local subtree is either a *store* operator, creating a file on a local disk, or a *send* operator. The *send* and *receive* operators shield the actual processing algorithms from the network, thus allowing to develop them as if they were meant for a centralized database system. Within a local subtree, processing is demand-driven, i.e., the root operator requests data from its child, which in turn requests input from its child or children. Typically, each demand results in one tuple, or, to be more exact, in a pointer to a tuple in main memory. The granularity of demands can change from tuple-at-a-time to page-at-a-time and back using special operators *pack* and *unpack*.

Before we describe the query processing algorithms to be implemented for the new database machine, we will outline a number of principles that we use as guidelines for algorithm design. In summary, we regard all resources as scarce: neither CPU cycles, main memory, I/O bandwidth, or network bandwidth can be considered plentiful and can be substituted freely for another scarce resource. There are several reasons for this approach. First, we do not know at this point which of these resources is most likely to be a bottleneck in a cube-connected database machine. Second, if one resource turns out to be the bottleneck for a particular cube size, system configuration, and usage pattern, a different resource may be the bottleneck in a different configuration. Third, in order to achieve the highest possible performance, we simply cannot afford to waste any resources. Let us review in some more detail the consequences of this approach.

Even though we expect to employ a relatively large number of fairly fast microprocessors, CPU cycles will be a scarce resource. In fact, we believe it possible that in a system with a disk attached to each processor, the system may become CPU bound. We will put the available processing power to its most effective use by employing the following principles. First, the algorithms must contain only a minimal number of synchronization points. Each synchronization point introduces the potential of one CPU waiting for another one, with obvious disadvantages. Second, the operating system kernel must not provide undesirable "services" [Stonebraker1981a]. In the context of DataCube, we consider virtual memory management and process preemption as particularly undesirable. Third, if multiple operators, e.g., *select*, *project*, *join*, are performed within a single CPU, we adopt a demand-driven dataflow model. Thus, synchronization of two operators within a single CPU is achieved with the cost of one procedure call. Fourth, we avoid memory-to-memory copying as much as possible, in particular at the interface between file system and relational operators, for temporary data structures such as hash tables, and for data transfer between operators. We will explore "page trading," i.e., avoiding copying between system buffers for used network communication and space belonging to relational operator processes by manipulating page table entries. Fifth, in order to reduce the effort for process

management, we will use "primed processes" [Gray1978a], i.e., processes will not disappear after performing a task but wait for a new processing request. Sixth, we will employ precompilation of predicates, projection lists, hash functions, etc., whenever possible. Finally, we will reduce the number of interrupts by using large disk pages (tracks) and network packets.

Main memory will be in high demand both as a disk cache and as work space for processing algorithms, namely hash tables. At each node, a memory manager allocates temporary work space to processes upon request, thus ensuring that no process retains unused work space. Second, the buffer manager accepts replacement hints. Since the buffer management code is executed very frequently, we found it imperative to use a very fast and simple algorithm. Our buffer manager distinguishes between clean and dirty pages and accepts hints whether to insert an unpinned page at the head or the tail of a LRU chain. Preliminary experiments show that these two distinctions bear significant impact on buffer hit ratios and replacement costs. More sophisticated algorithms, e.g., Hot Sets [Sacco1982a] and DBMIN [Chou1985a, Chou1985b], may be implemented and evaluated later.

Until proven otherwise, I/O bandwidth will be considered the scarcest resource in DataCube. Thus, we will provide indices for efficient selections, both ordered indices in the form of B-trees [Bayer1972a, Comer1979a] and unordered indices based on hashing [Fagin1979a]. Multi-dimensional indices [Robinson1981a, Guttman1984a, Nievergelt1984a] may be added later. In order to avoid sorting, trees can be used to cluster data files, or data files can be organized as trees. Furthermore, we intend to provide for record clustering for faster joins, i.e., tuples from different relations that logically belong together are stored in the same page. Finally, for large data files we will employ contiguous disk space allocation to reduce disk seek activity and read-ahead to overlap CPU and I/O activity.

Assessing the scarcity of network bandwidth is particularly difficult at this point as no cube-connected database machine has been implemented and evaluated to-date. In order to conserve network bandwidth, we are trying to reduce the number of messages between proces-

sors by using data-driven dataflow across the network, thus saving request messages, and by using local I/O only. Furthermore, simulation results in another dataflow database machine have shown that bursts of network activity between periods of network "silence" increase network contention drastically [Gerber1986a]. Thus, our algorithms are designed to result in evenly distributed network activity, employing data partitioning based on key ranges or on hashing. We will explore the potential for highly parallel communication, i.e., how the hardware capabilities of the cube-architecture can be used effectively by the database software. For increased parallelism, the optimizer will try to assign data partitions to processors such that the average communication path length is as short as possible.

Now that we have outlined the design principles for the database processing algorithms in DataCube, we will describe some of the algorithms in detail, and contrast them to earlier work where appropriate.

5. Database Processing Algorithms

5.1. Partitioning

The processing algorithms in DataCube partition data sets horizontally in order to achieve controlled and effective parallelism. Under horizontal partitioning we understand that a relation, be it a permanent database relation stored on disk or an intermediate result created during evaluation of a query, is divided into disjoint subsets of tuples called partitions. Typically, there is one partition per disk (for permanent relations) or per processing element (for intermediate results). In some cases, as will be discussed in the next sections, only a subset of the processors is used for a subquery, and fewer partitions will be used.

There are three partitioning algorithms that will be used. First, entry-sequenced or *round-robin* partitioning assigns the first page of data to the first disk in the system, the second page to the second disk, etc. Experiments in the GAMMA project demonstrated that granularity finer than a page can introduce significant overhead, and will not be considered for DataCube [Gerber1986a]. Second, *range-partitioning* uses a designated partitioning key, i.e., a data

attribute or a set of data attributes, and assigns a range to each disk drive in DataCube. Ranges can be assigned automatically or by explicit request. Clearly, choosing good range boundaries is important for load balancing and performance. Third, a randomizing or *hashing* function can be applied to a partitioning key, and the location is chosen depending on the hash value. This strategy is superior when a balanced load of tuples is required without knowledge of the key distribution, but may lead to disadvantages when data are required in some sort order for a later processing step or for presentation to the user.

Partitioning can be used for both stored relations and intermediate results. The implementation will use a fairly sophisticated *send* operator that determines for each tuple produced at a site (which can be the host processor when loading a relation) the next site of processing or storage using one of the partitioning mechanisms described above.

On disk, permanent relations can be clustered on a different key than the partitioning key. For example, if many queries refer to bank account tuples using an account number, partitioning on account number and building a secondary index on account number are probably the right choices. The clustered (primary) index can be on a different attribute, e.g., on customer name, thus providing fast retrieval of sorted sets, e.g., for a printed report. Distinguishing between partitioning key and clustering key has proven valuable in the GAMMA project [Gerber1986a].

5.2. Selections and Updates

Selections are performed using standard techniques like file scan, index scans, and read-ahead. DataCube's file system provides two kinds of indices, B-trees for domains with ordering and hash-based indices for domains without ordering. Multi-dimensional indices are currently node considered but may be added later. An index scan does not include lookup of the data records, it only retrieves the information associated with keys, e.g., record identifiers (RID's) of data records. Thus, we do not preclude using RID-oriented processing methods, e.g., intersection, union, or difference of RID lists for complex predicates or sorting RID lists before data

retrieval to reduce seeks. Retrieving data records from RID's is performed by the *functional join* operator which has to be included explicitly in a query tree by the optimizer.

For an equality or exact-match selection on a partitioning attribute, only one local index and/or file has to be examined. For range selections, only a subset of the disks need to be accessed. In these cases, the query packet contains information indicating which nodes with disk need to be activated, called the node selection bits.

As mentioned above, we consider it important that update transactions can be processed efficiently by DataCube. Three techniques will be used to ensure this. First, in order to avoid non-trivial data movement problems, we prohibit modifications to the partitioning attribute of permanent relations[2]. Thus, it is not necessary to redistribute tuples, to initiate processes to receive redistributed tuples, to readjust the partitioning ranges to ensure storage load balance, etc. Second, if an update query includes a predicate involving the partitioning key, only selected disk-nodes need to be activated. In the best case, only one node is needed for an update transaction. Third, we are currently exploring the use of multi-level transactions [Weikum1984a, Beeri1988a] in a multi-processor dataflow database machine.

For concurrency control, we intend to use local lock managers and local and global deadlock detection after lock time-outs. Write-ahead logs will be used for recovery purposes. Since the local disks cannot be used for the log, each disk node has one other disk node associated with it as log server. Log pages need to be assembled only once (at the originating site) and can be written to disk bypassing the buffer manager at the log server. The very short network latency realized in the latest generation of hypercube machines ensures minimal performance costs.

---

[2] Updates can still be performed by deleting a tuple and inserting a corrected tuple. In general, we expect that most relations have a suitable key that is not updated in normal operation, e.g., social security number. *Object identity* as defined in some database systems is a closely related, more general concept.

5.3. Join

For joins, we will implement four strategies. These are conventional merge join of sorted input relations, index lookup on hash indices, index lookup on tree indices, and hash join. The two index lookup methods will be implemented for both permanent and temporary ("on the fly") indices. Considering join methods using temporary indices requires implementation effort in the optimizer only, since the required functionality for building, scanning, and deleting indices is already included in the run-time system.

Hash join will probably be chosen most often by the query optimizer for the following three reasons. First, hash join is the fastest join method, provided that all necessary data structures can be kept in main memory. Second, since the two input relations are processed in consecutive phases, called the *build* and *probe* phases in GAMMA, bit vector filtering can be used to reduce the number of network transfers and hash table probes [Babb1979a, DeWitt1985a, DeWitt1986a]. Third, the hash table, presumed to be built on the smaller of the two relations before the larger relation is accessed, acts as a filter for the larger relation. Thus, tuples can be probed one by one against the hash table and output tuples can be produced in a continuous stream. Alternative hash join methods were analyzed in [DeWitt1984a] and one of them used very successfully in the GAMMA project [DeWitt1986a, Gerber1986a]. In some cases, however, the available main memory is not sufficient to hold all hash tables. Hash table overflow reduces performance significantly since it requires synchronization messages and a fair amount of data movement to and from possibly remote disks.

The input streams of a join operator are partitioned on the join attribute(s). Disjunct subsets of the join domain are assigned to the machines participating in the join called join nodes. In the general case, there are data streams from all machines producing join input to all join nodes. Some of these streams will be between physically adjacent nodes in the cube, whereas others will require a number of "hops," thus consuming a larger share of the point-to-point bandwidth. By analyzing the distributions of the join attributes, e.g., by using distribu-

tion information of a stored input relations [Piatetsky-Shapiro1984a, Graefe1987a], it will be possible to assign join key ranges to join nodes such that most of the communication paths are short. If communication bandwidth turns out to be a performance bottleneck, we expect a significant performance improvement by using a sophisticated node assignment. This is described in more detail on Section 6.

In contrast to GAMMA, DataCube will also use local joins for joins of stored relations on partitioning attributes. As an example, consider information about employees and their dependents which was normalized into an employee relation and a dependent relation. The employee relation contains a unique key SSN (social security number), and each tuple in the dependents relation contains a foreign key, the SSN of an employee. Many queries will include a join of the employee relation and the dependents relation. Instead of redistributing all tuples from both relations, DataCube will perform much better if the relations are partitioned using the SSN attributes using the same key ranges or hash function. If the relations have been partitioned in this favorable way, the join for all data stored on one disk can be performed on the CPU to which the disk is attached. In order to make use of all CPU's and all main memory in the system, near neighbors of each disk node can be used to assist in the join. In this case, data communication in the cube will be very localized, i.e., all data paths are very short, and many transfers can be performed simultaneously. Thus, smart partitioning of stored data can be used by the optimizer to improve performance.

In addition to local joins, we also intend to use physical links (pointers or record addresses) between data records on disk. On the user interface and data model level, we need to provide and enforce referential integrity or existence dependency constraints. These integrity constraints on the conceptual level allow to safely implement record pointers on the physical level. Notice that physical links only can be exploited successfully in multi-processor database machines with local disk drives if the partitioning strategies of related relations are organized in such a way that related records reside on the same node or disk drive.

Query optimization in the presence of physical links is only slightly more complex than in conventional query optimizers. A number of researchers have investigated relational user interfaces to network and hierarchical database systems and appropriate query optimization strategies, e.g., [Zaniolo1979a, Chen1984a, Rosenthal1985a]. We will use their insights and results when developing our query optimization component.

5.4. Duplicate Elimination and Aggregate Functions

The implementation of duplicate elimination and aggregate functions can be done in very similar ways. We plan to use a single operator with appropriate parameters. For aggregate functions, e.g., a request for "sums of salaries by department", a relation has to be partitioned into disjunct subsets of tuples with equal values for the attributes in the *by-list*. Within each subset, an aggregating function is performed. The results of the aggregate function are the by-list attributes of one of the tuples and an aggregated value. For duplication elimination, the by-list includes the entire tuple, and aggregating two tuples means simply deleting one.

In general, the entire input has to be consumed before the first result tuple can be determined, whether the implementation is based on sorting [Epstein1979a] or on hash tables [Gerber1986a]. Special cases exist for several of the standard aggregate functions. The case for "count $>$ 0" is well-recognized and implemented as *exist-clause* in some database systems. Other cases are bounds for "min", "max", and "sum" aggregate functions. In order to maintain the dataflow for duplicate elimination, the DataCube software will recognize and exploit these special cases of aggregate functions.

5.5. Division

Relational division is usually employed to express universal quantification or "for-all" clauses. For example, a request to find "the students who have taken all database classes," the database system must divide the transcript relation by the relation of all database classes. Relational division is not employed very often in current database applications, mainly because current implementations are rather slow. Most database systems do not implement division

directly. Instead, they transform a "for-all" clause into an expression with an aggregate function, e.g., find "the students who have taken as many database classes as there are database classes," and evaluate this query with methods based on sorting. We believe, however, that division could be a very important operator since it can be used to verify many integrity constraints. Furthermore, more intelligent user interfaces, e.g., based on logic programming, could greatly benefit from a database with efficient division.

We have developed a relational division operator based on hashing, using hash tables on the divisor (e.g., database classes) and on the quotient (e.g., students) and a number of bit maps. This algorithm performs very well in a centralized environment, and can easily be modified to operate in a multi-processor dataflow database machine [Graefe1988a]. In a distributed environment, two partitioning strategies can be used alone or together to distribute the processing load evenly over all processors in the database machine. Under *divisor partitioning*, the nodes determine quotient candidate tuples that must be filtered further in a final division step. Under *quotient partitioning*, the divisor must be broadcasted, whereas the dividend relation can be partitioned. Notice that the divisor is the much smaller relation, both in tuple width and in cardinality, and broadcasting it is probably not too significant. Also, notice that by embedding a tree in the hypercube structure, broadcast can be performed very efficiently. We expect that the cube architecture will greatly enhance the speed of broadcasting a divisor relation, thus supporting high performance relational division.

5.6. Sorting

Much work has been dedicated to parallel sorting, but only few algorithms have been implemented and evaluated in database settings, i.e., where the total amount of data is a large multiple of the total amount of main memory in the system. All such algorithms are variants of the well-known merge-sort technique and require a final centralized merge step [Bitton1984a]. In a highly parallel architecture, any centralized component that has to process all data is bound to be a severe bottleneck. Our research will use the duality between mergesort and

quicksort. Both algorithms are recursive divide-and-conquer algorithms. The difference is that mergesort first divides physically and then merges on logical keys, whereas quicksort first divides on logical keys and then combines physically by trivially appending sorted subarrays.

In general, one of the two phases dividing and combining is based on logical keys whereas the other arranges data item only physically. We call this the logical and the physical phase. Sorting algorithms for very large data sets stored on disk or tape are also based on dividing and combining. Usually, there are two distinct sub-algorithms, one for sorting within the main memory and one for managing subsets of the data set on disk or tape. The choices of which one is the logical phase and which one is the physical phase are independent for these two sub-algorithms. Among the main memory sort algorithms, mergesort uses physical dividing and logical combining, whereas quicksort uses logical dividing and physical combining. For practical reasons, e.g., ensuring that a run fits into main memory, the disk management algorithm typically uses physical dividing and logical combining (merging). A point of practical importance is the fan-in or degree of merging, but this is a parameter rather than a defining property of the algorithm.

For parallel sorting, we have essentially the same choices. Besides two choices described above for disk based sorts, a similar decision has to be made for the data exchange step. We assume that data redistribution among the processors is required, and that the cost of data transfer between processing nodes prohibits shipping a data item across the network more than once. We can perform the redistribution step either before or after local sort steps.

Imagine the latter method first. After all data have been sorted locally on all nodes, all sort-nodes start shipping their data with the lowest keys to the receiving node for this key range. The receiving node merges all incoming data streams, and is the bottleneck in this step, slowing down all other nodes. After this key range is exhausted on all machines, the receiving node for the second key range becomes the bottleneck, and so on. Thus, this method allows only for limited parallelism in the data exchange phase. The described problem can be alleviated by

reading all ranges in parallel. It is important, however, that a smart disk allocation strategy allows to do this without too many disks seek. We will explore the possible strategies and their implications on overall system performance.

On the other hand, starting the parallel sorting algorithm by dividing on logical keys at all data storage sites allows subsequent local sorting. Notice that, provided a sufficiently fast network in the first step, all data exchange can be done in parallel with no node depending on a single node for input values. First, all sites with data redistribute the data to all sites where the sorted data will reside. Second, all those sites which have received data sort them locally. This algorithm does neither contains a centralized bottleneck nor requires special disk allocation strategies, but it creates a new problem. The local sort effort is determined by the amount of data to be sorted locally. To achieve high parallelism in the local sort phase, it is imperative that the amount of data be balanced among the receiving processors. The amount of data at each receiving site is determined by the range of key values that the site is to receive and sort locally, and the number of data items with keys in this range. In order to balance the local sorting load, it is necessary to estimate the quantiles of the keys at all sites prior to the redistribution step. Quantiles are key values that are larger than a certain fraction of key values in the distribution, e.g., the median is the 50% or 0.5 quantile[3]. For load balancing among N processors, the i/N quantiles for i=1,...,N-1 need to be determined. Finding the median for a dataset distributed to a set of processors with local memory has been studied theoretically. We need to extend this research for a set of quantiles, and adapt it for a database setting, i.e., for disk-based large datasets. Furthermore, sufficient load balancing can probably be achieved using good estimates for the quantiles instead of the exact values. Our current work on describing data distributions using moments and density functions may provide significant assistance for this problem [Graefe1987a].

[3] Notice that if the distribution is "skewed", the mean and the median can differ significantly.

6. Query Optimization

The query optimizer for DataCube will be built using the EXODUS query optimizer generator [Graefe1987b, Graefe1987c]. In most of its functionality, the query optimizer will be similar to existing relational optimizers, e.g., [Selinger1979a]. At this point of time, our chief interest is in exploiting parallelism and the hardware interconnection structure of a hypercube for fast execution of relational operations.

In order to make best use of the available hardware structure, the network paths must be short, thus allowing for many parallel transmissions in the hypercube. Two techniques will be used. We discuss each of them in turn as they apply for the join operator. In modified form, they also apply for duplicate elimination and aggregate functions.

First, relations that are joined frequently are partitioned on the same key with the same partitioning rule. Most join operations are used to recombine relations that were decomposed when the database scheme was normalized [Maier1983a]. In such cases, foreign keys are used to maintain the association. Join operations on a foreign key and a "native" key can be performed locally. Thus, no network communication at all is necessary. In order to use the diskless cube nodes, a number of processors can be assigned to each local join, and each of the join operations is performed on a small subset of processors. If the disks are assigned to cube nodes with maximal distance, the subsets of nodes used for local joins can be disjunct and each subset is located close to the respective disk, resulting in short communication paths and parallel communication in the cube.

Second, if the key distribution of one or both input streams to a complex operator are known or can be anticipated, the optimizer tries to exploit this fact when assigning key ranges to processing nodes. Recall that for complex operators, in particular join, the data streams are partitioned into disjunct subsets which are assigned to processing nodes. Other researchers have chosen hash-partitioning, because it produces equal-sized subsets with good probability [DeWitt1986a]. However, it also results in indiscriminate usage of all logical communication

paths, including both physically long and short paths. In a hypercube, it is desirable to take advantage of physical proximity. For this reason, DataCube will support range partitioning for intermediate results. If the distribution before partitioning is known, e.g., for data partitioned on desired key or on a correlated key, key ranges can be assigned to processing nodes such that the average communication path length is minimized. We are currently working on tools to estimate data distributions that can be used for this purpose [Graefe1987a]. More details will be provided in forthcoming reports.

In a number of cases, queries cannot be satisfactorily optimized. We consider two cases. First, many queries embedded in application programs use a program variable in the query predicate. For example, the application program reads a value from the user's terminal and then uses it in a database selection predicate. For minimal run-time overhead, such queries are optimized when the application program is compiled [Chamberlin1981a]. However, the selectivity of the selection predicate cannot be reliably estimated during optimization, in particular if the program variable participates in a non-equality comparison ($\leq$, $<$, $>$, $\geq$). We propose to use *dynamic query evaluation plans* [Graefe1988b], which consist of more than one complete query evaluation plan and choose from among those at run-time using the actual values in the query predicate and a function that was prepared for this query and included in the plan by the query optimizer.

Second, with increasing complexity of a query, the reliability of selectivity estimates decreases since the estimation error multiplies over cascaded processing steps. For example, if the selectivity of each operation is estimated with an error bound of $\pm 25\%$, the result size of five cascaded operations is unpredictable for practical query optimization purposes. Thus, we will explore using a *stop-and-go* operator, suggested by DeWitt [DeWitt1988b]. This operator is essentially a scalar aggregate operation combined with a choose operation as used in dynamic query evaluation plans. While the implementation of this operator is easy and its potential benefit outstanding, it is currently not obvious to us when to insert this operator automatically

into a complex query evaluation plan.

7. Fault-Tolerance

In our first design, we do not intent to provide real fault tolerance. Instead, we will concentrate on providing transactions with synchronization atomicity and failure atomicity on both the compute and data server levels using traditional concurrency control and recovery techniques.

At this point, we only list some considerations how we plan to implement and eventually evaluate the integrated server architecture. First, we plan on using traditional locking and logging [Gray1978a]. Deadlocks are tested for after lock timeouts using a waits-for graph, first locally, then centrally with all single-site transactions omitted from the combined graph. Second, since we anticipate to attach only single disk drives to hypercube nodes, we need to introduce a system of ***buddy nodes***, i.e., pairs of nodes which serve as log server for each other. For fast undo of a single failed transaction, a new operation collects remotely all log records pertaining to a single transaction from their various log pages and sends them to the buddy node in as few network packets as possible. Third, a distributed signaling mechanism will be used to raise and process exceptions. Fourth, we are currently exploring how to use the nested transactions concept [Beeri1988a] to relate transactions on the two server levels. Finally, we will explore the use of the stop-and-go operator mentioned above for recovery and restart purposes.

8. Summary and Conclusions

In this paper, we have described the initial design and implementation plans for an integrated data and compute server, DataCube. Disk devices are attached to individual CPU's of a hypercube. The I/O software provides block or record oriented files, B-trees, hash indices, buffering. These additions enhance hypercube machines significantly and make them suitable for highly parallel database algorithms. In our design, each file is distributed over many disks, typically all, and can be partitioned by hashing or by range-partitioning. Processing algorithms

run distributedly on many CPU's, using dataflow techniques for data transfer between processes. The design exploits the very regular hardware structure to allow many concurrent data transfers simultaneously. The query optimizer takes data locations into account, giving preference to short communication paths. Data distributions, including correlations between attributes, are used to estimate intermediate result sizes and communication needs.

DataCube allows us to address a number of research issues. First, we will explore and evaluate the potential for sharing highly parallel hardware between two servers. Second, we will explore how to make best use of the parallel communication capabilities between servers. Third, we will experiment with and evaluate load balancing mechanisms and policies in this environment. Fourth, we will devise suitable changes to the query and data manipulation language and to the database query optimizer to control and exploit close interaction between the data and compute server subsystems.

After we have completed the design and implementation as described in this paper, we will continue to use DataCube for further research. First, we will evaluate the potential of hypercubes and the concept of distance between processing nodes and disks for database applications, including machines with very many nodes, and identify and remove performance bottlenecks. Second, we will experiment with novel query optimization techniques that delay carefully chosen strategy selections until run-time. Third, we will explore using this parallel architecture as a basis for a *design support* database system. We expect that physical record clustering and sophisticated buffer management for complex objects has more importance in this application domain and may require some redesign and reevaluation of policies.

References

Alexander1988a.
W. Alexander and G. Copeland, "Process and Dataflow Control in Distributed Data-Intensive Systems," *Proceedings of the ACM SIGMOD Conference*, pp. 90-98 (June 1988).

Babb1979a.
E. Babb, "Implementing a Relational Database by Means of Specialized Hardware," *ACM Transactions on Database Systems* **4**(1) pp. 1-29 (March 1979).

Baru1987a.
C.K. Baru and O. Frieder, "Implementing Relational Database Operations in a Cube-

Connected Multicomputer System," *Proceedings of the IEEE Conference on Data Engineering*, (February 1987).

Bayer1972a.
R. Bayer and E. McCreighton, "Organisation and Maintenance of Large Ordered Indices," *Acta Informatica* **1**(1972).

Beeri1988a.
C. Beeri, H.J. Schek, and G. Weikum, "Multi-Level Transaction Management," *Lecture Notes in Computer Science* **303** pp. 134-153 Springer Verlag, (April 1988).

Bitton1984a.
D. Bitton, D.J. DeWitt, D.K. Hsiao, and J. Menon, "A Taxonomy of Parallel Sorting," *ACM Computing Surveys* **16**(3) pp. 287-318 (September 1984).

Boral1983a.
H. Boral and D.J. DeWitt, "Database Machines: An Idea Whose Time Has Passed? A Critique of the Future of Database Machines," *Proceeding of the International Workshop on Database Machines*, Springer, (1983).

Bratbergsengen1984a.
K. Bratbergsengen, "Hashing Methods and Relational Algebra Operations," *Proceedings of the Conference on Very Large Data Bases*, pp. 323-333 (August 1984).

Chamberlin1981a.
D.D. Chamberlin, M.M. Astrahan, W.F. King, R.A. Lorie, J.W. Mehl, T.G. Price, M. Schkolnik, P. Griffiths Selinger, D.R. Slutz, B.W. Wade, and R.A. Yost, "Support for Repetitive Transactions and Ad Hoc Queries in System R," *ACM Transactions on Database Systems* **6**(1) pp. 70-94 (March 1981).

Chen1984a.
H. Chen and S.M. Kuck, "Combining Relational and Network Retrieval Methods," *Proceedings of the ACM SIGMOD Conference*, pp. 131-142 (June 1984).

Chou1985a.
H.T. Chou and D.J. DeWitt, "An Evaluation of Buffer Management Strategies for Relational Database Systems," *Proceedings of the Conference on Very Large Data Bases*, pp. 127-141 (August 1985).

Chou1985b.
H.T. Chou, "Buffer Management of Database Systems," *Ph.D. Thesis*, University of Wisconsin, (May 1985).

Codd1970a.
E.F. Codd, "A Relational Model of Data for Large Shared Data Banks," *Communications of the ACM* **13**(6) pp. 377-387 (June 1970).

Codd1972a.
E.F. Codd, "Relational Completeness of Database Sublanguages," pp. 65-98 in *Data Base Systems*, ed. R. Rustin,Prentice-Hall, New York (1972).

Comer1979a.
D. Comer, "The Ubiquitous B-Tree," *ACM Computing Surveys* **11**(2)(June 1979).

DeWitt1979a.
D.J. DeWitt, "DIRECT - A Multiprocessor Organization for Supporting Relational Database Management Systems," *IEEE Transactions on Computers* **28**(6) pp. 395-405 (June 1979).

DeWitt1984a.
D.J. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker, and D. Wood, "Implementation Techniques for Main Memory Database Systems," *Proceedings of the ACM SIGMOD*

Conference, pp. 1-8 (June 1984).

DeWitt1985a.
D.J. DeWitt and R.H. Gerber, "Multiprocessor Hash-Based Join Algorithms," *Proceedings of the Conference on Very Large Data Bases*, pp. 151-164 (August 1985).

DeWitt1986a.
D.J. DeWitt, R.H. Gerber, G. Graefe, M.L. Heytens, K.B. Kumar, and M. Muralikrishna, "GAMMA - A High Performance Dataflow Database Machine," *Proceedings of the Conference on Very Large Data Bases*, pp. 228-237 (August 1986).

DeWitt1988a.
D.J. DeWitt, S. Ghandeharizadeh, and D. Schneider, "A Performance Analysis of the GAMMA Database Machine," *Proceedings of the ACM SIGMOD Conference*, pp. 350-360 (June 1988).

DeWitt1988b.
D.J. DeWitt, *Personal Communication*. February 1988.

Epstein1979a.
R. Epstein, "Techniques for Processing of Aggregates in Relational Database Systems," *UCB/ERL Memorandum*, (M79/8)University of California, (February 1979).

Fagin1979a.
R. Fagin, J. Nievergelt, N. Pippenger, and H.R. Strong, "Extendible Hashing: A Fast Access Method for Dynamic Files," *ACM Transactions on Database Systems* **4**(3) pp. 315-344 (September 1979).

Gerber1986a.
R. Gerber, "Dataflow Query Processing using Multiprocessor Hash-Partitioned Algorithms," *Ph.D. Thesis*, University of Wisconsin, (October 1986).

Graefe1987a.
G. Graefe, "Selectivity Estimation Using Moments and Density Functions," *Oregon Graduate Center, Computer Science Technical Report*, (87-012)(November 1987).

Graefe1987b.
G. Graefe and D.J. DeWitt, "The EXODUS Optimizer Generator," *Proceedings of the ACM SIGMOD Conference*, pp. 160-171 (May 1987).

Graefe1987c.
G. Graefe, "Software Modularization with the EXODUS Optimizer Generator," *IEEE Database Engineering*, (December 1987).

Graefe1988a.
G. Graefe, "Relational Division: Four Algorithms and Their Performance," ***submitted for publication, also Oregon Graduate Center, Computer Science Technical Report***, (88-022)(June 1988).

Graefe1988b.
G. Graefe, "The Stability of Query Evaluation Plans and Dynamic Query Evaluation Plans," *Oregon Graduate Center, Computer Science Technical Report*, (88-003)(February 1988).

Gray1978a.
J.N. Gray, "Notes on Database Operating Systems," pp. 393-481 in *Operating Systems: An Advanced Course*, Springer, New York (1978).

Guttman1984a.
A. Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching," *Proceedings of the ACM SIGMOD Conference*, pp. 47-57 (June 1984).

Maier1983a.
D. Maier, *The Theory of Relational Databases,* Computer Science Press, Rockville, MD. (1983).

NCAR1987a.
NCAR, *Supercomputing: The View from NCAR,* National Center for Atmospheric Research, Scientific Computing Division, Boulder, CO. (December 1987).

Nievergelt1984a.
J. Nievergelt, H. Hinterberger, and K. C. Sevcik, "The Grid File: An Adaptable, Symmetric Multikey File Structure," *ACM Transactions on Database Systems* **9**(1) pp. 38-71 (March 1984).

Ozkarahan1977a.
E.A. Ozkarahan, S.A. Schuster, and K.C. Sevcik, "Performance Evaluation of a Relational Associative Processor," *ACM Transactions on Database Systems* **2**(2) pp. 175-195 (June 1977).

Piatetsky-Shapiro1984a.
G. Piatetsky-Shapiro and C. Connell, "Accurate Estimation of the Number of Tuples Satisfying a Condition," *Proceedings of the ACM SIGMOD Conference,* pp. 256-276 (June 1984).

Robinson1981a.
J.T. Robinson, "The K-D-B-Tree: A Search Structure For Large Multidimensional Dynamic Indices," *Proceedings of the ACM SIGMOD Conference,* pp. 10-18 (April-May 1981).

Rosenthal1985a.
A. Rosenthal and D.S. Reiner, "Querying Relational Views of Networks," pp. 109-124 in *Query Processing in Database Systems,* ed. W. Kim, D.S. Reiner, and D.S. Batory,Springer, Berlin (1985).

Sacco1982a.
G.M. Sacco and M. Schkolnik, "A Mechanism for Managing the Buffer Pool in a Relational Database System Using the Hot Set Model," *Proceeding of the Conference on Very Large Data Bases,* pp. 257-262 (September 1982).

Selinger1979a.
P. Griffiths Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, and T.G. Price, "Access Path Selection in a Relational Database Management System," *Proceedings of the ACM SIGMOD Conference,* pp. 23-34 (May-June 1979).

Stonebraker1981a.
M. Stonebraker, "Operating system support for database management," *Communications of the ACM* **24**(July 1981).

Stonebraker1987a.
M. Stonebraker, "The Design of the POSTGRES Storage System," *Proceeding of the Conference on Very Large Databases,* pp. 289-300 (August 1987).

Su1975a.
S.Y.W. Su and G.J. Lipovski, "CASSM: A Cellular System for Very Large Data Bases," *Proceedings of the Conference on Very Large Data Bases,* pp. 456-472 (1975).

Teradata1983a.
Teradata, *DBC/1012 Data Base Computer Concepts & Facilities.* 1983.

Weikum1984a.
G. Weikum and H.J. Schek, "Architectural Issues of Transaction Management in Multi-Layered Systems," *Proceedings of the Conference on Very Large Data Bases,* pp. 454-465 (August 1984).

Zaniolo1979a.
C. Zaniolo, "Design of Relational Views Over Network Schemas," *Proceedings of the ACM SIGMOD Conference*, pp. 179-190 (May-June 1979).