Distributed and Parallel Databases 1 (1993), 137–165

Parallel Database Systems: Open Problems and New Issues

PATRICK VALDURIEZ PATRICK.VALDURIEZ@INRIA.FR
Projet Rodin, INRIA, Rocquencourt, France

Received May 18, 1992, Revised August 18, 1992

Abstract. Parallel database systems attempt to exploit recent multiprocessor computer architectures in order to build high-performance and high-availability database servers at a much lower price than equivalent mainframe computers. Although there are commercial SQL-based products, a number of open problems hamper the full exploitation of the capabilities of parallel systems. These problems touch on issues ranging from those of parallel processing to distributed database management. Furthermore, it is still an open issue to decide which of the various architectures among shared-memory, shared-disk, and shared-nothing, is best for database management under various conditions. Finally, there are new issues raised by the introduction of higher functionality such as knowledge-based or object-oriented capabilities within a parallel database system.

Keywords: Parallel database systems, multiprocessor architectures, parallel database languages, data placement, query processing, parallel algorithms, rules, objects

1. Introduction

Database management and parallel processing technologies have evolved to a point that they can now be successfully combined to better support data-intensive applications. They are poised to take a central position in mainstream commercial information systems of the 1990s [78].

Commercial database technology has moved from the earlier hierarchical and network models to the relational model. The main advantages of relational database systems (RDBMSs) over their predecessors are data independence and high-level query languages (e.g., SQL). These advantages increase programmer productivity and favor automatic optimization. Furthermore, the set-oriented nature of the relational model facilitates distributed database management [56, 57]. Today, after a decade of optimization and tuning, RDBMSs can provide a performance level reaching that of earlier systems. Therefore, they are being extensively used in commercial data processing for decision-support or on-line transaction processing (OLTP) applications.

Parallel processing exploits multiprocessor computers to run application programs by using several processors cooperatively, in order to improve performance. Its prominent use is in scientific computing by improving the response time of numerical applications [47, 65]. The recent developments in both general-

purpose MIMD parallel computers using standard microprocessors and parallel programming techniques [55] will allow parallel processing to break into the data processing field.

The combination of database management and parallel processing is exemplified by the advances in *parallel database systems* [26]. These systems exploit recent multiprocessor computer architectures in order to build high-performance and high-availability database servers at a much lower price than equivalent mainframe computers. Note that performance was also the objective of the *database machines* (DBMs) in the 1970s and 1980s [42]. The problem faced by conventional database management has long been known as "I/O bottleneck" [13], induced by high disk access time with respect to main memory access time (typically hundreds thousands times faster). Initially, DBM designers tackled this problem through special-purpose hardware (e.g., by introducing data filtering devices within the disk). However, they failed because of a poor price/performance when compared to the software solution which can easily benefit from hardware progress in silicon technology [32]. A notable exception to these failures is the CAFS-ISP filter [7] which is bundled within ICL disk controllers for fast associative search and can be used by INGRES (when the optimizer decides to do so).

An important result of DBM research, however, is in the general solution to the I/O bottleneck. We can summarize this solution as *increasing the I/O bandwidth through parallelism*. For instance, if we store a database of size D on a single disk with throughput T, the system throughtput is bounded by T. On the contrary , if we partition the database across n disks, each with capacity D/n and throughput T' (hopefully equivalent to T), we get an ideal throughput of n^*T' which can be better consumed by multiple processors (ideally n). Note that the main memory database system solution [30] which tries to maintain the (active) database in stable main memory is complementary rather than alternative. In particular, the "memory access bottleneck" can also be tackled using parallelism in a similar way.

Therefore, parallel database system designers strive to develop software-oriented solutions in order to exploit multiprocessor hardware. The objectives of parallel database systems can be achieved by extending distributed database technology, for example, by partitioning the database across multiple (small) disks so that much inter- and intraquery parallelism can be obtained. This can lead to significant improvements in both response time and throughput (number of transactions per second). Motivated by set-oriented processing and application portability, most of the work in this area has focused on supporting SQL. There are already some relational database products that implement this approach, e.g., Teradata's DBC [54] and Tandem's NonStopSQL [72] and the number of such products will increase as the market for general-purpose parallel computers expands. In fact, there are already implementations of existing RDBMSs such as INGRES and ORACLE on parallel computers.

At first glance, the fact that there are successful commercial products may indicate that the important technical problems have been solved. On the contrary,

if one analyzes these systems carefully, it will be found that they typically rely on simple solutions (e.g., partitioning each relation across all nodes) and strong assumptions regarding the workload (e.g., debit-credit transactions of the TPC-B benchmark [38]). Open problems concern parallel system architectures, operating system support, data placement, parallel database programming languages, parallel algorithms, parallelizing compilation, and transaction management. They have been partially addressed in the context of distributed database systems [56] but are much more difficult because of the need to scale up to large numbers of components. Furthermore, it is still an open issue to decide which of the various architectures among shared-memory, shared-disk, and shared-nothing, is best for database management under various factors such as type of workload, application complexity and database size.

When applied to more complex application domains such as CAD/CAM, CASE, OIS, expert systems, etc., RDBMs show important limitations in terms of rule management, complex object support, and type system. To address these issues, two important next-generation DBMS technologies, namely knowledge bases and object-oriented databases, have emerged. Knowledge base systems (KBMSs) [33] should enable us to move from data management to more general knowledge management whereby knowledge can be captured within rules. Object-oriented database systems (OODBMSs) [86] try to combine object-oriented programming and database technologies in order to provide higher modelling power and flexibility to the application programmers. The higher functionality of KBMSs and OODBMSs make the performance issue far more sensitive than with RDBMSs and therefore raises new issues for implementing them on parallel computers.

In this paper, I critically review the parallel database system approach as the solution to high-performance and high-availability database management. The objectives are to exhibit the advantages and disadvantages of the various architectures and to present the open problems and new issues to be addressed by the research community in the near future.

The paper is organized as follows. Section 2 introduces the architectural aspects of parallel database systems and discusses the respective advantages and limitations of the three multiprocessor architectures along several important dimensions including the perspective of both end-users, database administrators and system developers. Section 3 discusses the open research problems. Section 4 concentrates on the new issues raised by next-generation parallel database systems.

2. Architectural considerations

A parallel database system can be loosely defined as a DBMS implemented on a tightly coupled multiprocessor. This definition excludes (distributed) DBMSs implemented on computer networks for they face specific problems such as geographical distribution, local autonomy, and heterogeneity [57] and do not face other problems due to large numbers of elements. However, this definition

does include many alternatives ranging from the straightforward porting of an existing RDBMS, which may require only rewriting the operating system interface routines, to a sophisticated combination of parallel processing and database system functions into a new hardware/software architecture. As always, we have the traditional trade-off between portability (to several platforms) and efficiency. I believe the sophisticated approach is better able to fully exploit the opportunities offered by a multiprocessor at the expense of portability. Interestingly, this gives different advantages to computer manufacturers and software vendors. It is therefore important to characterize the main points in the space of alternative parallel system architectures. In order to do so, I will make precise the parallel database system solution and the necessary functions. This will be useful in comparing the three basic parallel database system architectures.

2.1. Parallel database system solution

Before reading about the solution, a fair question the reader may ask is: "What is the problem? Is that problem important? and to whom?" Answering these questions requires looking at a global picture of our computerized society. Today, in a competitive world, enterprises of all kinds use and depend on timely available, up-to-date information. Information volumes are growing 25–35% per year and the traditional transaction rate has been forecast to grow by a factor of 10 over the next five years—twice the current trend in mainframe growth [29]. In addition, there is already an increasing number of transactions arising from computer systems in business-to-business interworking and by intelligent terminals in the home, office or factory.

The profile of the transaction load is also changing as decision-support queries, typically complex, are added to the existing simpler, largely clerical workloads. Thus, complex queries such as those macro-generated by decision support systems or system-generated as in production control will increase to demand significant throughput with acceptable response times. In addition, very complex queries on very large databases, generated by skilled staff workers or expert systems, may hurt throughput while demanding good response times.

From a database point of view, the problem is to come up with database servers that support all these types of queries efficiently on possibly very large on-line databases. However, the impressive silicon technology improvements alone cannot keep pace with these increasing requirements. Microprocessor performance is now increasing 50% per year, and memory chips are increasing in capacity by a factor of 16 every six years. RISC processors today can deliver between 50 and 100 MIPS (the new 64 bit DEC Alpha processor is predicted to deliver 200 MIPS at cruise speed!) at a much lower price/MIPS than mainframe processors. This is in contrast to much slower progress in disk technology which has been improving by a factor of 2 in response time and throughput over the last 10 years. With such progress, the I/O bottleneck worsens with time.

The solution is therefore to use large-scale parallelism to magnify the raw power of individual components by integrating these in a complete system along with the appropriate parallel database software. Using standard hardware components is essential to exploit the continuing technology improvements with minimal delay. Then, the database software can exploit the three forms of parallelism inherent in data-intensive application workloads. *Interquery parallelism* enables the parallel execution of multiple queries generated by concurrent transactions. *Intraquery parallelism* makes the parallel execution of multiple, independent operations (e.g., select operations) possible within the same query. Both interquery and intraquery parallelism can be obtained by using *data partitioning*. Finally, with *intraoperation parallelism*, the same operation can be executed as many suboperations using *function partitioning* in addition to data partitioning. The set-oriented mode of database languages (e.g., SQL) provides many opportunities for intraoperation parallelism. For example, the performance of the join operation can be increased significantly by parallelism [25, 80].

2.2. *Functional architecture*

A parallel database system acts as a *server* for multiple *client* computers in the now common client-server organization in computer networks. The client typically embeds application-specific software such as graphical interfaces, DBMS front-end tools such as 4GLs, and client-server interface software. It can run on virtually anything from a personal computer or workstation to a mainframe. The parallel database system supports the database functions and the client-server interface, and possibly general-purpose functions. The latter capability distinguishes a parallel database system from a database machine which is fully dedicated to database management and cannot, for instance, run a C program written by a user. To limit the potential communication overhead between client and server, a high-level powerful interface (set-at-a-time rather than record-at-a-time) that encourages data-intensive processing by the server is necessary.

This approach naturally extends to the more general distributed database approach with multiple servers, each acting as a local site in the network. What is needed then is an additional software layer at each server to provide distribution transparency. Because this layer can be clearly separated from the parallel database management functions, I will ignore it for simplicity in the rest of the paper.

Ideally, a parallel database system should provide the following advantages with a much better price/performance than its mainframe counterparts.

High performance. This can be obtained through several complementary solutions: database-oriented operating system support, parallelism, optimization, and load balancing. Having the operating system constrained and "aware" of the specific database requirements (e.g., buffer management) simplifies the implementation of

low-level database functions and therefore decreases their cost. For instance, the cost of a message can be significantly reduced to a few hundred of instructions by specializing the communication protocol. This solution has been exploited in the early database machines like the IDM [76]. Parallelism can increase throughput (using interquery parallelism) and decrease transaction response times (using intraquery and intraoperation parallelism). However, decreasing the response time of a complex query through large-scale parallelism may well increase its total time (by additional communication) and hurt throughput as a side effect. Therefore, it is crucial to optimize and parallelize queries in order to minimize the overhead of parallelism, e.g., by constraining the degree of parallelism for the query. Load balancing is the ability of the system to divide a given workload equally among all processors. Depending on the multiprocessor architecture, it can be achieved by static physical database design or dynamically at run-time.

High-availability. Because a parallel database system consists of many similar components, it can exploit data replication to increase database availability. Thus, in the event of a disk failure, the copy of the data may still be available on one or more disks at no additional cost (unlike log-based recovery). However, replica support requires the implementation of control protocols that enforce copy consistency. The most used protocol is ROWA (read one, write all) which converts a logical read operation to a physical read operation on any one of the copies, but a logical write operation is translated into physical writes on all copies. In a highly parallel system with many small disks, the probability of a disk failure at anytime can be higher (than in an equivalent mainframe). Therefore, it is essential that a disk failure does not imbalance the load, e.g., by doubling the load on the available copy. Solutions to this problem require partitioning copies in such a way that they can also be accessed in parallel [43].

Extensibility. In a parallel environment, accommodating increasing database sizes or increasing performance demands (e.g., throughput) should be easier. Extensibility is the ability of smooth expansion of the system by adding processing and storage power to the system. Ideally, the parallel database system should demonstrate two advantages [26]: *linear scaleup* and *linear speedup*. Linear scaleup refers to a sustained performance for a linear increase in both database size and processing and storage power. Linear speedup refers to a linear increase in performance for a constant database size and linear increase in processing and storage power. Furthermore, extending the system should require minimal reorganization of the existing database.

Assuming a client-server architecture, the functions supported by a parallel database system can be divided into three subsystems much like in a typical RDBMS. The differences, though, have to do with implementation of these functions which must now deal with parallelism, data partitioning and replication, and distributed transactions. Depending on the architecture, a processor can support all (or a subset) of these subsystems. Figure 1 shows the architecture using

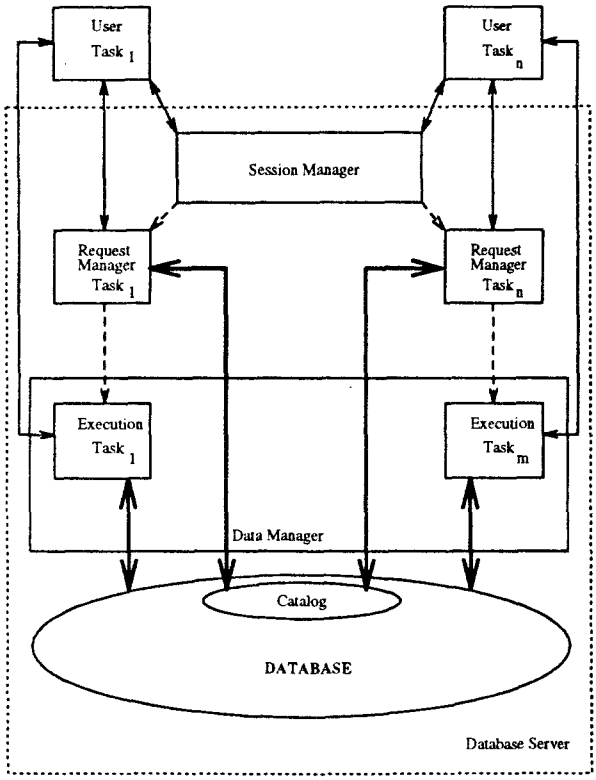

Figure 1. General architecture of a parallel database system.

these subsystems named after [9]. Solid double arrows indicate communication, bold double arrows indicate data access, and dotted arrows indicate task creation.

Session manager. The session manager plays the role of a transaction monitor (like TUXEDO [3]), providing support for client interactions with the server. In particular, it performs the connections and disconnections between the client processes and the two other subsystems. Therefore, it initiates and closes user sessions (which may contain multiple transactions). In case of OLTP sessions, the session manager is able to trigger the execution of pre-loaded transaction code within data manager modules.

Request manager. The request manager receives client requests related to query compilation and execution. It can access the catalog which holds all meta-information about data and programs. The catalog itself should be managed as a database in the server. Depending on the request, it activates the various

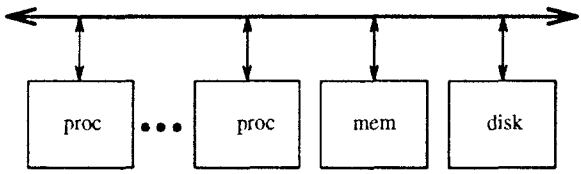

Figure 2. Shared-memory architecture.

compilation phases, triggers query execution and returns the results as well as error codes to the client application. Because it supervises transaction execution and commit, it may trigger the recovery procedure in case of transaction failure. To speed up query execution, it may optimize and parallelize the query at compile-time.

Data manager. The data manager provides all the low-level functions needed to run compiled queries in parallel, i.e., database operation execution, parallel transaction support, cache management, etc. If the request manager is able to compile dataflow control, then synchronization and communication among data manager modules is possible. Otherwise, transaction control and synchronization must be done by a request manager module.

2.3. *Parallel system architectures*

A parallel system represents a compromise in design choices in order to provide the aforementioned advantages with a better cost/performance. One guiding design decision is the way hardware components, i.e., processors, memories, and disks, are interconnected through some fast communication medium. Parallel system architectures range between two extremes, the *shared-memory* and the *shared-nothing* architectures, and a useful intermediate point is the *shared-disk* architecture [61].

2.3.1. Shared-memory. In the shared-memory approach (see Figure 2), any processor has access to any memory module or disk unit through a fast interconnect (e.g., a high-speed bus or a crossbar switch). Several new mainframe designs such as the IBM3090 or Bull's DPS8, and symmetric multiprocessors such as Sequent and Encore follow this approach.

Examples of shared-memory parallel database systems include XPRS [69], DBS3 [9], and Volcano [36], as well as portings of major RDBMSs on shared-memory multiprocessors. In a sense, the implementation of DB2 on an IBM3090 with six processors [20] was the first example. All the shared-memory commercial products today exploit interquery parallelism only (i.e., no intraquery parallelism).

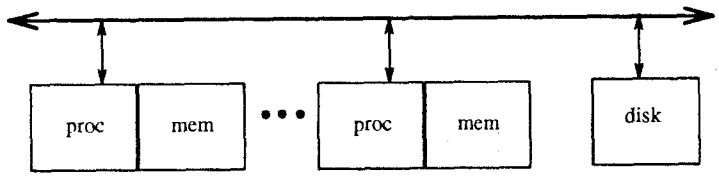

Figure 3. Shared-disk architecture.

Shared-memory has two strong advantages: simplicity and load balancing. Since meta-information (catalog) and control information (e.g., lock table) can be shared by all processors, writing database software is not very different than for single-processor computers. In particular, interquery parallelism comes for free. Intraquery parallelism requires some parallelization but remains rather simple. Load balancing is excellent since the system assigns tasks to processors at run-time based on the actual load.

Shared-memory has three problems: cost, limited extensibility, and low availability. High cost is incurred by the interconnect which is fairly complex because of the need to link each processor to each memory module or disk. With faster and faster processors (even with larger caches), conflicting accesses to the shared-memory increase rapidly and degrade performance [74]. Therefore, extensibility is limited to tens of processors (20 on a Sequent on Encore). Finally, since the memory space is shared by all processors, a memory fault may effect most processors thereby hurting database availability. A solution is to duplex memory as in Sequoia systems.

2.3.2. Shared-disk. In the shared-disk approach (see Figure 3), any processor has access to any disk unit through the interconnect but exclusive (nonshared) access to its main memory. Then, each processor can access database pages on the shared disk and copy them into its own cache. To avoid conflicting accesses to the same pages, global locking and protocols for the maintenance of cache coherency are needed [52].

Examples of shared-disk parallel database systems include IBM's IMS/VS Data Sharing product and DEC's VAX DBMS and Rdb products. The implementation of ORACLE on DEC's VAX cluster and NCUBE computers is also using the shared-disk approach since it requires minimal extensions of the RDBMS kernel. Note that all these systems exploit interquery parallelism only.

Shared-disk has a number of advantages: cost, extensibility, load balancing, availability, and easy migration from uniprocessor systems. The cost of the interconnect is significantly less than with shared-memory since standard bus technology may be used. Given that each processor has enough cache memory, interference on the shared disk can be minimized. Thus, extensibility can be better (hundreds of processors). Load balancing can be as good as with shared-

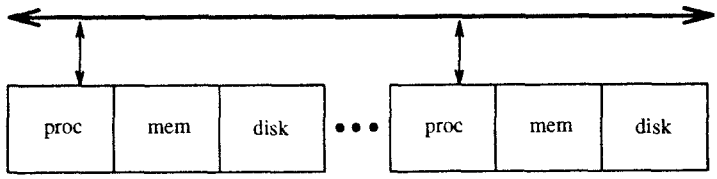

Figure 4. Shared-nothing architecture.

memory for the same reasons. Since memory faults can be isolated from other processor-memory nodes, availability can be higher. Finally, migrating from a centralized system to shared disk is relatively straightforward since the data on disk need not be reorganized.

Shared-disk suffers from higher complexity and potential performance problems. It requires distributed database system protocols, such as distributed locking and two-phase commit which can be complex [52]. Furthermore, maintaining the coherency of the copies can incur high communication overhead among the nodes. Finally, access to the shared disk is a potential bottleneck.

*2.3.3. **Shared-nothing.*** In the shared-nothing approach (see Figure 4), each processor has exclusive access to its main memory and disk unit(s). Then, each node can be viewed as a local site (with its own database and software) in a distributed database system. Therefore, most solutions designed for distributed databases such as database fragmentation, distributed transaction management and distributed query processing may be reused.

Examples of shared-nothing parallel database systems include the Teradata's DBC and Tandem's NonStopSQL products as well as a number of prototypes such as Bubba [12], Eds [29], Gamma [28], Grace [31], Prisma [5], and Arbre [50]. All these systems exploit both inter- and intraquery parallelism.

As demonstrated by the existing products, e.g., [73], shared nothing has three main virtues: cost, extensibility, and availability. The cost advantage is the same as for shared disk. By implementing a distributed database design which favors the smooth incremental growth of the system by the addition of new nodes, extensibility can be better (thousands of nodes). For instance, Teradata's DBC can accommodate 1024 processors. With careful partitioning of the data on multiple disks, linear speedup and linear scaleup could be achieved for simple workloads. By replicating data on multiple nodes, high availability can be also achieved.

Shared-nothing suffers also from higher complexity, in addition to load balancing problems. Higher complexity is due to the necessary implementation of distributed database functions assuming large numbers of nodes. Load balancing is more difficult to achieve because it relies on the effectiveness of database partitioning for the query workloads. Unlike shared-memory and shared-disk,

load balancing is decided based on data location and not the actual load of the system. Furthermore, the addition of new nodes in the system presumably requires reorganizing the database to deal with the load balancing issues.

2.3.4. Comparisons. Let us briefly compare these alternative design approaches based on their potential advantages (high-performance, high-availability, and extensibility). It is fair to say that, for a small configuration (e.g., less than 20 processors), shared-memory can provide the highest performance because of better load balancing. Shared-disk and shared-nothing, however, outperform shared memory in terms of availability and extensibility. Finally, shared-nothing can scale up to higher numbers of processors than shared memory and shared disk.

Thus, it appears that shared-nothing is the only choice for high-end systems (e.g., requiring more than thousands of TPS of the TPC-B benchmark). However for small-to-medium systems (e.g., requiring less than 1000 TPS), shared-memory and shared-disk are interesting alternatives [10].

3. Open research problems

As we have seen in the previous section, there are representative products for each parallel system architecture. One aspect common to the most recent products is the single focus on SQL and relational databases for business data processing applications. Although technology transfer from research into products has been impressive, there are still research problems which hamper to fully exploit the range of possibilities offered by multiprocessor computers. In this section, I discuss the major open problems which have to do with architectures, data placement, parallel database languages, and parallel query processing.

3.1. Architectural aspects

Although the respective advantages and limitations of each architectural model for data management are now well understood, this is not so for hybrid architectures. An example of hybrid architecture is one where some disk or memory modules are shared and some others are not. Furthermore, there are important problems with respect to the use of disk arrays [60], operating system support, and internetworking of database servers. A final related consideration which we discuss below is the need for comprehensive benchmarks.

Most of the research problems have been investigated assuming a given architectural paradigm, typically shared-memory or shared-nothing. This assumption is generally motivated by a strong faith of the designers in the chosen architecture and their will to concentrate on software solutions (possibly as a reaction to the failure of hardware-oriented DBMs). Furthermore, this approach simplifies

implementation somewhat. A shared-memory design can simply extend a single-processor DBMS design with run-time parallelization [69] while a shared-nothing design can reuse and extend distributed database techniques.

However, some hybrid parallel system architectures may be better. Given the limited extensibility of shared-memory and the load balancing problem of shared-nothing, an interesting compromise is to have a shared-nothing system in which each node is itself a shared-memory multiprocessor. Having a few powerful nodes in a shared-nothing architecture also simplifies the data placement problem. Then the question is whether to be extensible and scalable to a limited number of very powerful shared-memory nodes or to a higher number of less powerful nodes. The Encore 93 series follows the first approach by allowing several shared-memory nodes with up to 32 processors to be connected through a high-speed network. Teradata's P90 [17] (potential successor of the DBC) follows the second approach by targeting the interconnection of up to 512 nodes, each being a shared-memory four-processor board, using a fast tree-structured bus (the BYnet).

In addition, considering the trends in supercomputer architectures, we can imagine parallel database systems with processors of different speeds (and prices). One advantage of such architectures is that the inherently sequential tasks (which hurt throughput) could be sent to the faster processors. For instance, the request manager component which essentially does multipass compilation and optimization is a good candidate for the faster processors. More experimental study is needed to decide the best architecture and configuration for different workloads. The work pioneered in [10, 74] is a good starting point.

Disk arrays are being considered as a promising approach to high-performance I/O architectures. A disk array consists of a large number of small, inexpensive disks and a high-bandwidth interconnect (of hundreds of megabytes per second) for disk-memory transfers. Therefore, it can provide very high throughput by exploiting I/O parallelism and thus reduce the I/O bottleneck. It can also provide high availability through replication. Data placement techniques designed for shared-nothing systems (partitioning) can be reused for disk arrays [84]. If disk arrays are successful (which is yet to be proven), an interesting issue is their integration in a parallel database system. From the outside, a disk array is a black box with its own complex software (some disk array controllers consist of 1 million lines of C code). While it may be easy to use disk arrays with shared memory or shared disk, it is a hard, if not hopeless, problem for shared nothing. A shared-nothing design can be viewed as a disk array with memory and processing power attached to each disk. Therefore, there is a potential design mismatch between the two. However, Teradata's P90 intends to use a number of disk arrays, each attached to two four-processor nodes. More research is definitely needed to understand the potential advantages of disk arrays in shared-nothing systems.

As for any dedicated system, specific operating system support for parallel data management can be very cost-effective. Two approaches can be applied. The

first one creates a brand new dedicated operating system almost from scratch, e.g., the Bubba operating system, which implements a single-level store where all data are uniformly represented in a virtual address space [22]. Although this approach can lead to the best performance, it restricts the use of the parallel system to database operations and cannot, for instance, run C or Cobol programs. The second approach tries to capitalize on modern, operating system microkernels such as Chorus [62] or Mach [44], and extends it in a way that can provide efficient support for database-oriented functions. In this case, the database-oriented operating system is just a subsystem as UNIX can be. Thus, it is more open to support non database applications as well.

If parallel data servers become prevalent, it is not difficult to see an environment where many of them are placed on a backbone network. This gives rise to distributed systems consisting of clusters of processors [37]. An interesting concern in such an environment is internetworking. Specifically, the execution of database queries which span multiple, and possibly heterogeneous, clusters creates at least the problems of distributed multidatabase systems. However, there are the additional problems that the queries have to be optimized not only for execution in parallel on a cluster of servers, but also for execution across a network.

Ultimately, the comparison of alternative parallel database system architectures will require specific benchmarks. Benchmarking is the only impartial way of assessing the performance/price of a system for a given workload. There are now standard benchmarks for DBMS and transaction processing systems [38] which stem from major research efforts, e.g., the TPC [4], Wisconsin [11], or Engineering database benchmarks [18]. However, most benchmarks measure an isolated aspect of a system. TPC measures the throughput of a workload of simple (debit-credit) transactions whereas the Wisconsin benchmark measures the response time of complex (decision-support) queries. AS^{3}AP [75], however, does include mixed workloads including simple and complex transactions as well as utilities (e.g., database load). For parallel database systems, more work is needed to come up with benchmarks which can stress linear speedup and linear scaleup under mixed workloads including simple and complex transactions as well as batch programs.

3.2. Data placement

In a parallel database system, proper data placement is essential for load balancing. Ideally, interference between concurrent, parallel operations can be avoided by having each operation to work on an independent dataset. These independent datasets can be obtained by *declustering* (horizontal partitioning) of the relations based on a function (hash function or range index) applied to some placement attribute(s), and allocating each partition on a different disk. Similar to horizontal fragmentation in distributed databases, declustering is useful to obtain interquery

parallelism, by having independent queries working on different partitions, and intraquery parallelism, by having a query's operations working on different partitions. As for clustering, declustering can be single-attribute or multiattribute. In the latter case [35], an exact match query requiring the equality of multiattributes can be processed by a single node without communication. The choice between hashing or range index for partitioning is a design issue: hashing incurs less storage overhead but provides direct support for exact-match queries only, while range index can also support range queries. Initially proposed for shared-nothing systems, declustering has been shown to be useful for shared-memory designs as well, by reducing memory access conflicts [9].

Full declustering, whereby each relation is partitioned across all the nodes, causes problems for small relations or systems with large numbers of nodes. A better solution is *variable declustering* where each relation is stored on a certain number of nodes as a function of the relation size and access frequency [21]. This can be combined with multirelation clustering to avoid the communication overhead of binary operations.

When the criteria used for data placement change to the extent that load balancing degrades significantly, dynamic reorganization should be performed. An important issue is to perform such dynamic reorganization on-line (without stopping the incoming of transactions) and efficiently (through parallelism). By contrast, existing database systems perform static reorganization for database tuning [66]. Furthermore, reorganization should remain transparent to compiled programs that run on the parallel system. In particular, programs should not be recompiled because of reorganization. Therefore, the compiled programs should remain independent of data location. This implies that the optimizer does not know the actual disk nodes where a relation is stored or where an operation will actually take place. The set of nodes where a relation is stored, when a certain operation is to be executed, is called its *home*. Similarly, the set of nodes where the operation will be executed is called the home of the operation. However, the optimizer needs abstract knowledge of the homes (e.g., relation R is hashed on A over 20 nodes) and the run-time system makes the association between the home and the actual nodes.

A serious problem in data placement is dealing with skewed data distributions which may lead to nonuniform partitioning and hurt load balancing. Hybrid architectures with nodes of different memory and processing power can be exploited usefully here. Another solution is to treat nonuniform partitions appropriately, e.g., by further declustering large partitions. The separation between logical and physical nodes is also useful since a logical node may correspond to several physical nodes.

A final complicating factor is data replication for high availability. The naive solution is to maintain two copies of the same data, a primary and a backup copy, on two separate nodes. However, in case of a node failure, the load of the node having the copy may double, thereby hurting load balancing. To avoid this problem, several high-availability data replication strategies have been proposed

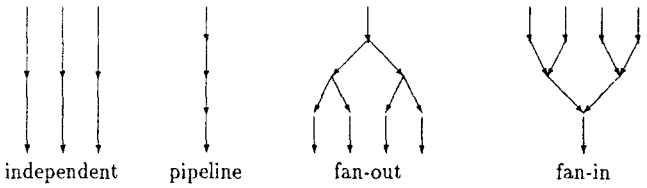

Figure 5. Types of parallelism.

and recently compared [43]. An interesting solution is Teradata's interleaved declustering which declusters the backup copy on a number of nodes. In failure mode, the load of the primary copy gets balanced among the backup copy nodes. However, reconstructing the primary copy from its separate backup copies may be costly. In normal mode, maintaining copy consistency may also be costly. A better solution is Gamma's chained declustering which stores the primary and backup copy on two adjacent nodes. In failure mode, the load of the failed node and the backup nodes are balanced among all remaining nodes by using both primary and backup copy nodes. In addition, maintaining copy consistency is cheaper. An open issue remains to perform data placement taking into account data replication. Similar to the fragment allocation in distributed databases, this should be considered an optimization problem.

3.3. Parallel database languages

There are various forms of parallelism. Figure 5 shows four simple kinds of parallelism graphically.

A few key ideas can be derived from applying these parallelism structures to problems in intensive data processing:

- Dividing problems is the essence of parallelism. Division into independent subproblems gives independent parallelism, while dividing into incremental computations gives pipeline parallelism. Set mappings naturally adapt to independent parallelism (the same instruction is independently applied to each element of a set) while stream mappings adapt to pipeline parallelism (some instructions are successively applied to each element of a stream). Thus, sets and streams suggest a divide-and-conquer format for specifying mappings which is implicitly also a format for specifying parallelism.
- Divide-and-conquer computations can be represented by combining these types of parallelism. "Dividing" a problem is represented by fan-out nodes in the graph, while conquering gathers results into a set (with independent parallelism), a stream (with pipeline parallelism), and/or an aggregate

(with fan-in parallelism). Thus, divide-and-conquer solutions of problems naturally capture these kinds of parallelism.

- Relational algebra operators can often be naturally expressed as divide-and-conquer computations.

These ideas raise hope for a parallel data processing system that rests upon divide-and-conquer techniques. However, such a system must deal with several technical issues to be viable.

A first problem is that the relational model offers no way to talk about order among data (e.g., sorted relations, or ordered tuples). Relational languages are therefore inadequate for specifying "stream processing," in which ordered sequences of data are processed sequentially [58]. Hence, streams cannot be exploited to specify pipeline parallelism following a data-parallelism paradigm. Pipeline parallelism is generally used, transparently to the user, in lower-level languages implementing relational algebra (e.g., PLERA [19] or PFAD [40]).

A second problem is that parallel data processing requires effective data partitioning capabilities. Typically, a relational query (select-project-join expression) is translated into a low-level form of relational algebra with explicit (low-level) parallel constructs. Data partitioning is used to spread the computation of relational algebra operators among parallel processors. This partitioning is typically defined during the physical database design and then exploited by a compiler. Most of the time, a partitioned computation requires that processors exchange intermediate results in order to compute the final result.

Ideally, data partitioning must be expressible by the programmer or the compiler within a parallel database language. This is essential to automatically extract parallelism and lead to efficient implementations on parallel database systems. Specifying parallel computations over relations often requires specifying how data partitioning (fan-out parallelism) will be done and how distributed results will be collected (fan-in parallelism). Database models have been developed before that permit expression of both ordering among tuples and data partitioning. For example, the FAD language of Bubba has operators that express various forms of fan-out and fan-in parallelism [23]. FAD is a strongly typed set-oriented database language based on functional programming and relational algebra. It provides a fixed set of higher-order functions to aggregate functions, like the `pump` parametrized aggregate operator and the grouping operator. The `pump` operator applies a unary function to each element of a set, producing an intermediate set which is then "reduced" to a single datum using a binary function that combines the intermediate set elements. Indeed, `pump` naturally expresses a special case of fan-out and fan-in parallelism. At the same time, the `group` operator permits set partitioning.

The SVP model [59] goes one step further in allowing sets, streams and parallelism to be captured in a unified framework formalizing divide-and-conquer mappings. SVP models collections, whose specialization leads to sets or streams, as series-parallel graphs which ease expressing parallel data processing. An

important class of queries called *transducers* generalizes aggregate operations and set or stream operations. They lead to high-level specification of independent and pipeline parallelism. Thus, SVP is a possible formal foundation for further research in parallel data programming languages.

3.4. Parallel query processing

Parallel query processing refers to the automatic translation of a query, expressed with a centralized execution model in mind, into an efficient execution plan, and its parallel execution. Such translation has two important aspects. First, the translation must be a correct transformation of the input query so that the execution plan actually produces the expected result. The formal basis for this task is transformation rules associated with relational algebra operators. Second, the execution plan must be *optimal* in that it minimizes a cost function that captures resource consumption. This requires investigating equivalent alternative processing trees in order to select the best one. These tasks are more or less difficult depending on whether they are performed at compile-time or run-time. Finally, the execution plan must be loaded for execution in the parallel system and run with concurrent transactions. We can divide the processing of a query in three steps: optimization, parallelization and execution. Each step faces specific issues that I address below.

Declustered data placement is the basis for the parallel execution of database queries. Therefore, much work has been devoted to the design of parallel algorithms which exploit such placement. However, more work is still needed to handle skewed data [85]. By dividing each set-oriented operation in a SIMD fashion, much intraoperation parallelism can be exploited. The basic principle is "to execute where the data is," that is, exploit data placement as much as possible by sending operations to their data. However, it is sometimes better to dynamically rearrange a relation to increase parallelism if the overhead of reorganization is less than its benefit. This is more likely to apply to intermediate relations than possibly large base relations.

The parallelization of an operation is based on a global and a local algorithm. The global algorithm decomposes the operation into local ones, to which another algorithm is applied. For instance, given a relation declustered across n nodes, the operation Select(R) is equivalent to the union of n operations Select(R_i), with $i = 1, \ldots, n$, where each individual operation can be done in parallel. However, if the select predicate involves the placement attributes, fewer nodes than n (ideally one) need be involved.

Parallelizing binary operations is more involved since, for optimal parallelism, it requires each operand relation to be declustered the same way. For example, if R and S are both declustered across n nodes using the same function on the join attribute, the operation Join(R, S) is equivalent to the union of n parallel operations Join(R_i, S_i), with $i = 1, \ldots, n$. Parallel join algorithms in [25,

67, 80] attempt to make such condition available by reorganizing the relations if necessary. Hashing has became the major technique for parallelizing binary operations such as join, union, and difference. Order-preserving hashing can also be used for parallel sorting. However, determining uniform ranges of attribute values to be handled by each processor is critical for load balancing [46]. Sampling looks like a promising solution for parallel sorting and, more generally, to deal with skewed data [27].

Given the existence of parallel algorithms, the importance of run-time parameters, such as processors load, raises the issue of static versus dynamic parallelization. In centralized DBMS, query optimization is performed prior to the execution of the query, hence called static, for two reasons. First, it can be done within a compiler, thereby reducing run-time optimization cost. Second, it can better exploit knowledge regarding physical schema and data placement. In parallel database systems, static optimization can still be beneficial but is made difficult by a larger search space, a more complex cost model and possibly high optimization cost. The search space is larger because of the wide range of parallel execution strategies. For instance, bushy processing trees should be considered for they can provide a higher degree of parallelism than linear trees.

The cost model provides the necessary abstraction of the parallel execution system in terms of access method cost functions, and an abstraction of the database in terms of physical schema information and related statistics. A number of important restrictions are often associated with the cost model, limiting the effectiveness of optimization. It is a weighted combination of cost components such as I/O, CPU, and communication and can capture either response time (RT) or total time (TT). Although TT optimization may increase throughput by minimizing resource consumption, RT optimization may well hurt throughput because of the overhead of parallelism. A potentially beneficial direction of research is to apply multiple query optimization [63] whereby a set of important queries from the same workload are optimized together. This would provide opportunities for load balancing and for exploiting common intermediate results. Other problems are the accuracy of the cost functions for parallel algorithms and the impact of update queries on throughput. Careful analysis of the cost functions should provide insights for determining useful heuristics to cut down the number of alternative execution plans.

There is a necessary trade-off between optimization cost and quality of the generated execution plans. High optimization costs are unacceptable for ad hoc queries which are executed only once. Therefore, it is critical to study the application of efficient search strategies that avoid the exhaustive search approach. More important, a different search strategy should be used depending on the kind of query (simple versus complex) and the application requirements (ad hoc versus repetitive). This requires support for controllable search strategies [48]. An interesting other solution is to perform optimization itself in parallel.

Static optimization can be followed by static parallelization, which translates the optimal execution plan into a parallel program. This approach is used in

Bubba and DBS3, and relies on a parallel database programming language. For instance, DBS3's PLERA [19] supports operators for local execution, data transfer and execution control. This approach allows decentralized control of the parallel program and offers control optimization opportunities. However, to achieve load balancing, there are some decisions which should be made at run-time, e.g., allocation of physical processors. Generating code to make such decisions is not easy. Furthermore, as database languages get increasingly powerful, we need more complex rules for performing correct transformations from centralized to parallel programs.

Dynamic parallelization is used in XPRS to select the optimal degree of parallelism for the operations based on the actual run-time load of the system. This approach is fairly simple. Optimization is done by a centralized query optimizer and the sequential execution plan is parallelized at run-time. Thus, excellent load balancing can be achieved. However, potentially better execution plans, e.g., bushy trees, are de facto ignored by the centralized optimizer. More work is needed to better combine the advantages of static and dynamic parallelization.

Parallel execution of (compiled) queries has to deal with the problems of transaction, initiation and transaction scheduling. Transaction termination faces the issues of distributed transactions, i.e., the cost of the commit and replica protocols. Transaction initiation involves loading code and starting-up processes. This function is trickier in shared nothing since it requires code to be shipped across nodes. In [2, 39], several activation mechanisms are proposed and compared. For ad hoc queries, piggybacking code fragments with the data shipped or callback for the code are useful. For precompiled queries, dynamic activation of preloaded code is generally superior.

Transaction scheduling is difficult in the case of mixed workloads comprising short on-line transactions and decision-support queries. The latter ones tend to acquire large numbers of locks at the expense of short transactions, and therefore hurt throughput. The practical solution duplicates the database so the on-line database is accessed by the short transactions and a snapshot database by the decision-support queries. To support such mixed workloads on the same database, solutions such as versioning [61, 69] need further investigation.

4. Next-generation parallel database systems

The penetration of database technology into new application areas with different requirements than traditional business data processing has motivated the notion of *next-generation database systems* [16, 68, 71]. One major objective is that the data model to be supported must be more powerful than the relational model, without compromising its advantages (data independence and high-level query languages). When applied to more complex application domains such as engineering, office information systems, and expert systems, the relational data

model exhibits limitations in terms of rule management, type system and complex object support. To address these issues, two important technologies, KBMSs and OODBMSs, are currently being investigated. Initially considered antagonistic, many believe today that a combination of their capabilities into *deductive and object-oriented database* (DOOD) systems will shape next-generation, universal database systems. For the same reasons which led to parallel relational database systems, implementing KBMSs and OODBMSs on parallel computers can be cost-effective. Obviously, this presents new, challenging research problems in addition to the current issues of KBMSs and OODBMSs.

4.1. Parallel KBMS

KBMSs should enable us to move from data management to more general knowledge management by abstracting the reasoning mechanism from the application programs and encapsulating it within the DBMS. RDBMSs typically provide a limited form of knowledge support through assertions and views. KBMSs (also called deductive database systems) manage and process possibly complex rules against large amounts of data (also called facts) within the DBMS rather than within a separate subsystem. Rules can be declarative (assertions) or imperative (triggers). By isolating the application knowledge and behavior within rules, KBMSs provide control over knowledge which can be better shared among users. Furthermore, the high expressive power of rule programs aids application development. These advantages imply increased programmer productivity and application performance. Because it is based on first-order logic, deductive database technology subsumes relational database technology.

We can isolate two alternative approaches to KBMS design. The first one extends a relational DBMS with a more powerful rule-based language (e.g., RDL [45] and ESQL [34]), while the second approach extends first-order logic into a declarative programming language such as Datalog [77] or LDL [53]. The two approaches raise similar issues, some of which have been partially addressed by the logic programming community, typically with strong assumptions such as a small, single-user database.

Rule management, as investigated in deductive databases, is essential since it provides a uniform paradigm to deal with semantic integrity control, views, protection, deduction and triggers. Much of the work in deductive databases has concentrated on the semantics of rule programs and on processing deductive queries, particularly in the presence of recursive and negated predicates [8]. However, there are a number of open issues related to the enforcement of the semantic consistency of the knowledge base, optimization of rule programs involving large amounts of facts and rules, integration of rule-based languages with other tools (e.g., application generators), and providing appropriate debugging tools.

Implementing a KBMS on a parallel system can capitalize on relational database technology, e.g., by storing facts within relations. Therefore, the new issues have more to do with the processing of parallel deductive queries than with parallel data management. The major technique for deductive query processing is *bottom-up evaluation* which starts from the facts and applies the rules necessary to derive the answer to the query. The bottom-up processing of a deductive query has two major steps. First, the query is merged with the relevant rules, the ones which use the query predicates. The parameter bindings given in the query are propagated in the rule bodies. This step produces a rule program with bindings. Second, the rule program is translated into an optimized program in the internal database language, e.g., an extension of relational algebra with control constructs such as "while do" and parallel constructs, which can be subsequently executed by the parallel system.

The rapid access to the relevant rules in the first step can be achieved using some form of index, typically a *predicate connection graph* [51]. If the rule base is large, then an interesting solution is to use declustering to favor parallel rule access. Then the problem is to find partitioning functions which can cluster the connected subgraphs. Another problem is with triggers which are rules fired as results of updates or other events. Including potential firing of triggers within compiled queries may be practically infeasible since triggers can recursively call other triggers. A possibility is to implement run-time rule firing based on updates as an extension of the transaction management mechanism [70]. One difficulty is to be able to access trigger information from the updated data. In a shared-nothing architecture, this may well increase communication overhead if triggers refer to nonlocal data. This suggests that data placement and preloading of trigger code be addressed together.

Parallel deductive query processing is made difficult by the presence of additional capabilities, such as recursive rules, and the larger range of parallel execution strategies for such capabilities. As pioneered in [82], most of the work in this area has focused on extending query processing to support the *transitive closure* of declustered relations in parallel. The transitive closure operator is essential to solve data-intensive problems, such as the *bill-of-material* (finding the number of elements connected to a given part). Parallel algorithms can be very effective in exploiting the regularity of the data to be processed. Promising techniques include hash-based partitioning [82], extensions of direct techniques [1] and semantic-based data partitioning [15]. The latter technique partitions the relation graph into disconnected sets but is essentially static, i.e., updates to the graph may imply repartitioning. More general Datalog programs have also been recently considered for parallel execution [15] using data partitioning. Much more work is still needed to improve the existing algorithms and provide a general framework to process parallel deductive queries. Such a framework will be also of interest for analyzing and comparing the performance of various techniques.

4.2. Parallel OODBMS

Object-oriented databases combine object-oriented programming (OOP) and database technologies in order to provide higher modeling power and reduce the chronic mismatch between databases and programming languages. I see three important classes of OODBMS (see representative systems in [16]). The first one extends the relational model and SQL with OO capabilities, e.g., Postgres and Starbust. The second class is *persistent OOP* which extends an OOP language, e.g., C++ or Smalltalk, with database capabilities, e.g., Ontos and ObjectStore. The last class relies on a new, semantic data model, e.g., O2, which combines OO and database features. Each OODBMS seems to have its respective advantages and weaknesses as well as its niche market (such a discussion is beyond the scope of this paper). However, OODBMSs, and more generally DOOD systems, are a first step toward ubiquitous systems for the construction of multiparadigmal applications with persistent objects which capture all the enterprise's data [6]. In addition to traditional database functions, the primary functions to be supported are abstract data types (with method code), type inheritance, type safety, and complex objects.

Over the last years, OODBs have been the subject of intensive research and experimentation. However, the theory and practice of developing parallel OODBMSs have yet to be fully developed. Even though some of the solutions developed for relational systems are applicable, the high degree of generality introduced by the OODB data models creates significant difficulties. In this section, I review the more important issues related to the overall system architecture, object management, operating system support, transaction processing, and query models and processing.

OODB applications typically arise in workstation-server environments. To better exploit the increasing MIPS and memory power of the workstations, it then makes sense to shift some of the functionality from the server to the client workstations. In [24], several alternative architectures are proposed and compared: object-server, page-server and file-server. In the object-server architecture (smart server), the server services requests for objects access and update, with centralized locking and logging of objects. Most RDBMSs and OODBMSs (e.g., Ontos) follow this approach. In the page-server architecture (dumb server), the server services page demands and page updates, with centralized locking and logging of pages. O2 and ObjectStore follow this approach. For simplicity, I ignore the file-server approach which, for the sake of this paper, is comparable to page-server. The bottom line is that page-server outperforms object-server when the workstation has a large buffer pool and data accesses show good locality of reference and the opposite is true otherwise. An additional point is that page-server forces page-level locking and may not be appropriate for applications with much multiuser concurrency (for which object-level locking is better). In their conclusion, the authors suggest that a hybrid architecture where pages (or

files) are read but objects are written back may be best, although more difficult. More work is definitely needed to study alternative hybrid architectures.

These results are very influential when looking at the viability and research issues of implementing an OODBMS server on a parallel system. Clearly, the pure page-server approach does not seem to raise new issues, except perhaps for the declustering of sets of pages containing complex objects. Fortunately (or unfortunately depending on your perspective), this approach is hardly compatible with the support of deductive capabilities as required by DOOD systems. For instance, complex rules for semantic integrity, e.g., triggers, can be more efficiently enforced close to the data, i.e., by the server [66]. In the case of object-server or hybrid architectures, we have all the issues at the cross-roads of OODBMSs and parallel database systems that I discuss below.

Object management in a parallel system is most difficult in the case of shared-nothing architectures since we have the issues of distributed object management [57]. With shared-disk or shared-memory architectures, one could use the traditional OODBMS solutions for object clustering. Efficient management of objects with complex connections is difficult. When objects can be hierarchical and contain possibly shared subobjects, object clustering and indexing is a major issue. With object identity and object sharing (the ability of an object to be referenced by multiple parents), garbage collection of objects is problematic. Furthermore, large-size, multimedia objects such as graphics and images with their associated methods need special attention.

Distributed object management should rely on a storage model which can capture the clustering and declustering of complex objects. Solutions developed for relational systems can be applied to collections of objects, i.e., top-level objects. However, the main problems remain the support of global object identity and object sharing. An interesting avenue of research is *uniform object management* [22] which provides a uniform treatment of objects regardless of whether they are transient versus persistent, local versus nonlocal, or memory resident versus disk resident. This can be achieved efficiently using a single-level store where all objects are represented in a virtual address space. Bubba implements a single-level store per node. With 64-bit processors, it should be easier to implement distributed single-level stores where the entire database is mapped in distributed virtual memory space. However, distributed garbage collection is a difficult problem [64].

The development of parallel (or distributed) OODBMSs bring to the forefront the issues related to proper operating system support. The issues are more interesting in this case since the development of object-oriented distributed operating systems has also been studied independently. Object-oriented technology can serve as the common platform to eliminate the impedance mismatch between the programming languages, database management systems, and operating systems. The integration of the first two have been addressed by OODBMS designers. However, the integration of OODBMSs and object-oriented operating systems have not yet been studied and remains an interesting research

issue. One problem in using object-orientation to bring these systems together is their differing understandings of what an object is and their quite different type systems. Nevertheless, if the next-generation database systems are to exhibit an easier cooperation with the operating systems than today's RDBMSs do, these issues need to be addressed.

Difficulties are introduced in transaction management for three reasons. First, objects can be complex thereby making variable-granularity locking essential. Second, support for dynamic schema evolution requires efficient solutions for updating schema data. Third, to address the different requirements of the targeted application domains, several concurrency control algorithms need be supported (e.g., pessimistic and optimistic concurrency control). Furthermore, engineering applications typically require specific support for cooperative transactions or long-duration nested transactions. In object-oriented systems, full generality is typically required such that complex transactions operate on complex object structures. Furthermore, the object model may treat transactions as first-class objects, both adding complexity and more opportunities to support multiple transaction types in one system [14]. Thus, in a parallel OODBMS, the issue of mixing various workloads is even more difficult than in parallel relational database systems.

In order not to compromise the obvious advantages of relational systems, an OODBMS ought to provide a high-level query language for object manipulation. While there has been some proposals for calculus and algebras for OODBs, query optimization remains an open issue. OODB queries are more complicated and can include path traversals and ADT operations as part of their predicates. The initial work on OODB query processing does not consider object distribution and parallelism. The efficient parallel processing of OODB queries can borrow from distributed relational query processing to exploit the declustering of collection objects. However, achieving correct program transformations is more involved due to the higher expressive power of the query languages. To reuse most of the technology developed for parallel database systems, a promising approach is to clearly separate the search space, the search strategies and the parallel cost model of the optimizer. In [49], path expressions are viewed as implicit joins, presumably more efficient than explicit value-based joins, e.g., by a combination of object identifiers and join indices [79]. This makes it possible to include them with ADT operations in complex (recursive) queries.

5. Conclusion

Parallel database systems strive to exploit modern multiprocessor architectures using software-oriented solutions for data management. Their promises are high-performance, high-availability and extensibility with a much lower price/performance ratio than their mainframe counterparts. Furthermore, parallelism is the only viable solution for supporting very large (terabyte) databases within a single system.

Although there are successful commercial SQL-based products, a number of open problems hamper the full exploitation of the capabilities for parallel systems. These problems touch on issues ranging from those of parallel processing to distributed database management. The first open issue is to decide which of the various architectures among shared-memory, shared-disk, and shared-nothing, is best for database management. For a small configuration (tens of processors), shared-memory can provide the highest performance because of better load balancing. Shared-disk and shared-nothing, however, outperform shared-memory in terms of availability and extensibility. On the other hand, shared-nothing can scale up to higher numbers of processors. Thus, it appears that shared-nothing is the only choice for high-end systems. But for small-to-medium systems, shared-memory and shared-disk are interesting, simpler alternatives.

Interesting compromises can be obtained from hybrid architectures, e.g., a shared-nothing system in which each node is itself a shared-memory multiprocessor. Then the question is whether to be extensible and scalable to a limited number of very powerful shared-memory nodes or to a higher number of less powerful nodes. The possibility of using disk arrays makes the question more difficult. Besides these architectural considerations, the following issues require more work:

1. Operating system support for efficient parallel data management with openness to nondatabase applications as well, e.g., using microkernel operating system technology.
2. Benchmarks to stress linear speedup and linear scaleup under mixed workloads including simple and complex transactions as well as batch programs.
3. Declustered data placement techniques to deal with skewed data distributions and data replication so as to achieve load balancing, including in failure mode.
4. Parallel data processing languages that rest upon divide-and-conquer techniques to specify independent and pipeline parallelism in a high-level way.
5. Parallel query processing with cost-based optimization and automatic parallelization to deal with mixed workloads of precompiled transactions and complex ad hoc queries.

The introduction of higher functionality, such as knowledge-based or object-oriented capabilities, within a parallel database system also raises new issues. To support knowledge-based capabilities, data placement and parallel query processing must be significantly revised to deal with possibly large rule bases and complex deductive queries. The introduction of object-oriented capabilities also creates significant difficulties related to complex object declustering, transaction management, proper object-oriented operating system support and parallel processing of object-oriented queries. Finally, the integration of the two capabilities into more powerful DOOD systems for the support of multi-paradigmal applications poses other challenging problems.

Acknowledgments

I wish to thank Stott Parker for helping on the section on parallel database languages, and David DeWitt, Jim Gray, Tamer Özsu, Dennis Shasha, and the anonymous referees for useful comments on earlier versions of this paper. This work was done in cooperation with Bull System Products, Research and Advanced Development within the IDEA Esprit Project.

References

1. R. Agrawal and H. Jagadish, "Multiprocessor transitive closure algorithms," in *Int. Symp. Databases in Parallel and Distributed Systems*, Austin, Texas, 1988.
2. W. Alexander and G. Copeland, "Process and dataflow control in distributed data-intensive systems," in *ACM SIGMOD Int. Conf.*, Chicago, 1988.
3. J. Andrade, M. Carges, and K. Kovach, "Building a transaction processing system on UNIX system," in *Unix Transaction Processing Workshop*, Pittsburgh, 1989.
4. Anon et al., "Measure of transaction processing power," *Datamation*, April 1985.
5. P. Apers et al., "Prisma/DB: a parallel main-memory relational DBMS," *IEEE Trans. Data Knowledge Engg.* (to appear).
6. M. Atkinson, "A vision of persistent systems," in *Int. Conf. Deductive and Object-Oriented Databases*, Munich, 1991.
7. E. Babb, "Implementing a relational database by means of specialized hardware," *ACM Trans. Database Systems*, vol. 4, no. 1, 1979.
8. F. Bancilhon and R. Ramakrishnan, "An amateur's introduction to recursive query processing strategies," in *ACM SIGMOD Int. Conf. Management of Data*, Washington, DC, 1986.
9. B. Bergsten, M. Couprie, and P. Valduriez, "Prototyping DBS3, a shared-memory parallel database system," in *Int. Conf. Parallel and Distributed Information Systems*, Miami, 1991.
10. A. Bhide and M. Stonebraker, "Performance comparison of two architectures for fast transaction processing," in *Int. Conf. Data Engineering*, Los Angeles, 1988.
11. D. Bitton, D. DeWitt, and C. Turbyfill, "Benchmarking database systems: a systematic approach," in *Int. Conf. VLDB*, Florence, Italy, 1983.
12. H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez, "Prototyping bubba, a highly parallel database system," *IEEE Trans. Knowledge Data Engg.*, vol. 2, no. 1, 1990.
13. H. Boral and D.J. DeWitt, "Database machines: an idea whose time has passed? A critique of the future of database machines," in *Int. Workshop on Database Machines*, Munich, 1983.
14. A. Buchmann, M.T. Özsu, M. Hornick, D. Georgakopoulos, and F.A. Manola, "A transaction model for active distributed object systems," in A. Elmagarmid (ed.), in *Transaction Models for Advanced Database Applications*, Morgan Kaufmann, 1992.
15. F. Cacace, S. Ceri, and M. Houtsma, "A survey of parallel execution strategies for transitive closure and logic programs," Technical Report No. 923, University of Twente, The Netherlands, 1990.
16. Special Issue on Next-Generation Database Systems, *Comm. ACM*, vol. 34, no. 10, 1991.
17. F. Carino and P. Kostamaa, "Exegesis of DBC/1012 and P-90– industrial supercomputer database machines," in *Parallel Architectures and Languages Europe*, Paris, 1992.
18. R.G.G. Cattell and J. Skeen, "Object operations benchmark," *ACM Trans. Database Systems*, vol. 17, no. 1, 1992.
19. C. Chachaty, P. Borla-Salamet, and M. Ward, "An approach for the design of a parallel query language," in *Parallel Architectures and Languages Europe*, Paris, 1992.

20. J. Cheng et al., "IBM database 2 performance: design, implementation and tuning," *IBM Syst. J.*, vol. 23, no. 2, 1984.
21. G. Copeland, W. Alexander, E. Boughter, and T. Keller, "Data placement in Bubba," in *ACM SIGMOD Int. Conf.*, Chicago, 1988.
22. G. Copeland, M. Franklin, and G. Weikum, "Uniform object management," in *Int. Conf. on EDBT*, Venice, 1990.
23. S. Danforth and P. Valduriez, "A FAD for data-intensive applications," *IEEE Trans. Data Knowledge Engg.*, vol. 4, no. 1, 1992.
24. D.J. DeWitt, P. Futtersack, D. Maier, and F. Velez, "Study of three alternative workstation-server architectures for object-oriented database systems," in *Int. Conf. VLDB*, Brisbane, Australia, 1990.
25. D.J. Dewitt and R. Gerber, "Multiprocessor join algorithms," in *Int. Conf. VLDB*, Stockholm, 1985.
26. D.J. Dewitt and J. Gray, "Parallel database systems: the future of high performance database systems," *Comm. ACM*, vol. 35, no. 6, 1992.
27. D.J. Dewitt, J.F. Naughton, D.A. Schneider, "Parallel sorting on a shared-nothing architecture using probabilistic splitting," in *Int. Conf. Parallel and Distributed Information Systems*, Miami, 1991.
28. D.J. Dewitt et al., "The GAMMA database machine project," IEEE *Trans. Knoweledge Data Engg.*, vol. 2, no. 1, 1990.
29. EDS Database Group, "EDS-collaborating for a high-performance parallel relational database," in *ESPRIT Conf.*, Brussels, 1990.
30. M. Eich, "Main memory database research directions," in *Int. Workshop Database Machines*, Deauville, 1989.
31. S. Fushimi, M. Kitsuregawa, and H. Tanaka, "An overview of the system software of a parallel relational database machine GRACE," in *Int. Conf. VLDB*, Kyoto, 1986.
32. S. Gamerman and M. Scholl, "Hardware versus software filtering: the VERSO experience," in *Int. Workshop Database Machines*, Grand Bahama Island, 1985.
33. G. Gardarin and P. Valduriez, *Relational Databases and Knowledge Bases*, Addison-Wesley: Reading, MA, 1990.
34. G. Gardarin and P. Valduriez, "ESQL2: an extended SQL2 with F-logic semantics," in *IEEE Int. Conf. Data Engineering*, Phoenix, 1992.
35. S. Ghandeharizadeh, D. DeWitt, and W. Qureshi, "A performance analysis of alternative multi-attributed declustering strategies," in *ACM SIGMOD Int. Conf.*, San Diego, 1992.
36. G. Graefe, "Encapsulation of parallelism in the volcano query processing systems," in *ACM SIGMOD Int. Conf.*, Atlantic City, 1990.
37. J. Gray, "Transparency in its place—the case against transparent access to geographically distributed data," in *Int. Conf. Distributed Computing Systems*, Paris, 1990.
38. J. Gray (ed.), *The Benchmark Handbook for Database and Transaction Processing Systems*, Morgan Kaufman, 1991.
39. G. Hallmark, "Function request shipping in a database machine environment," in *Int. Workshop on Database Machines*, Deauville, 1989.
40. B. Hart, S. Danforth, and P. Valduriez, "Parallelizing FAD: a database programming language," in *Int. Symp. Databases in Parallel and Distributed Systems*, Austin, 1988.
41. W. Hong and M. Stonebraker, "Optimization of parallel query execution plans in XPRS," in *Int. Conf. Parallel and Distributed Information Systems*, Miami, 1991.
42. D. Hsiao (ed.), *Advanced Database Machine Architectures*, Prentice Hall, 1983.
43. H.-I. Hsiao and D. DeWitt, "A performance study of three high-availability data replication strategies," in *Int. Conf. Parallel and Distributed Information Systems*, Miami, 1991.
44. M.B. Jones and R.F. Rashid, "Mach and matchmaker: kernel and language support for object-oriented distributed systems," in *Int. Conf. OOPSLA*, Portland, Oregon, 1986.
45. J. Kiernan, C. de Maindreville, and E. Simon, "Making deductive databases a practical reality: a step forward," in *ACM SIGMOD Int. Conf.*, Atlantic City, 1990.

46. M. Kitsuregawa and Y. Ogawa, "A new parallel hash join method with robustness for data skew in super database computer (SDC)," in *Int. Conf. VLDB,* Brisbane, Australia, 1990.
47. J.S. Kowalik (ed.), *Parallel MIMD Computation: The HEP Supercomputer and Its Applications*, MIT Press: Cambridge, MA, 1985.
48. R.S.G. Lanzelotte and P. Valduriez, "Extending the search strategy in a query optimizer," in *Int. Conf. Very Large Data Bases*, Barcelona, Spain, 1991.
49. R.S.G. Lanzelotte, P. Valduriez, and M. Zait, "Optimization of object-oriented recursive queries using cost-controlled strategies," in *ACM SIGMOD Int. Conf.*, San Diego, 1992.
50. R. Lorie et al., "Adding intra-parallelism to an existing DBMS: early experience," *IEEE Bull. Database Engg.*, vol. 12, no. 1, 1989.
51. D. McKay and S. Shapiro, "Using active connection graphs for reasoning with recursive rules," in *Int. Joint Conf. AI*, Vancouver, Canada, 1981.
52. C. Mohan and I. Narang, "Efficient locking and caching of data in the multi-system shared disks transaction environment," IBM Research Report RJ 8301, 1991.
53. S. Naqvi and S. Tsur, *A Logical Language for Data and Knowledge Bases,* Computer Science Press, New York, 1989.
54. P.M. Neches, "The anatomy of a database computer," Digest of Papers, *COMPCON*, San Francisco, 1985.
55. A. Osterhaug, *Guide to Parallel Programming on Sequent Computer Systems*, Prentice Hall: Englewood Cliffs, 1989.
56. T. Özsu and P. Valduriez, *Principles of Distributed Dababase Systems*, Prentice Hall: Englewood Cliffs, 1991.
57. T. Özsu and P. Valduriez, "Distributed databases: where are we now?," *IEEE Comput.*, vol. 24, no. 8, 1991.
58. D.S. Parker, "Stream data analysis in Prolog," in L. Sterling (ed.), *The Practice of Prolog*, MIT Press, 1990.
59. S. Parker, E. Simon, and P. Valduriez, "SVP, a data model capturing sets, streams and parallelism," in *Int. Conf. VLDB,* Vancouver, 1992.
60. D. Patterson, G. Gibson, and R. Katz, "A case for redundant arrays of inexpensive disks," in *ACM SIGMOD Int. Conf.*, Chicago, 1988.
61. H. Pirahesh et al., "Parallelism in RDBMS: architectural issues and design," in *Int. Symp. Databases in Distributed and Parallel Systems,* Dublin, 1988.
62. M. Rozier et al., "Chorus distributed operating systems," *Comput. Systems*, vol. 1, no. 4, 1988.
63. T. K. Sellis, "Multiple query optimization," *ACM Trans. Database Systems*, vol. 13, no. 1, 1988.
64. M. Shapiro, O. Gruber, and D. Plainfossé, "A garbage detection protocol for a realistic distributed object-support system," INRIA Research Report No. 1320, Rocquencourt, France, 1990.
65. J.A. Sharp, *An Introduction to Distributed and Parallel Processing*, Blackwell Scientific Publications: Oxford, 1987.
66. D. Shasha, *Database Tuning: a Principled Approach*, Prentice Hall: Englewood Cliffs, NJ, 1992.
67. D. Shasha and T.L. Wang, "Optimizing equijoin queries in distributed databases where relations are hash partitioned," *ACM Trans. Database Systems*, vol. 16, no. 2, 1991.
68. A. Silberschatz, M. Stonebraker, and J.D. Ullman (eds.), "Database systems: achievements and opportunities," in *PJ Report of the NSF Invitational Workshop on the Future of Database Systems Research*, Technical Report TR-90-22, UT, Austin, 1990.
69. M. Stonebraker et al., "The design of XPRS," in *Int. Conf. VLDB*, Los Angeles, 1988.
70. M. Stonebraker, L.A. Rowe, and M. Hiroshama, "The implementation of POSTGRES," *IEEE Trans. Knowledge Data Engg.*, vol. 2, no. 1, 1990.
71. M. Stonebraker, L.A. Rowe, B. Lindsay, J. Gray, M. Carey, M. Brodie, P. Bernstein, and D. Beech, "Third-generation data base system manifesto," *ACM SIGMOD Record*, vol. 19, no. 3, 1990.
72. The Tandem Database Group, "NonStop SQL – a Distributed high-performance, High-Availability Implementation of SQL," in *Int. Workshop High-Performance Transaction Systems*, Asilomar, CA, 1987.

73. The Tandem Database Group, "A benchmark of NonStop SQL on the debit credit transaction," in *ACM SIGMOD Int. Conf.*, Chicago, 1988.
74. S. Thakkar and M. Sweiger, "Performance of an OLTP application on symmetry multiprocessor system," in *Int. Symp. Computer Architecture*, Seattle, 1990.
75. C. Turbyfill, C. Orji, and D. Bitton, "AS^3 AP: an ANSI SQL standard scalable and portable benchmark for relational database systems," in J. Gray (ed.), *The Benchmark Handbook for Database and Transactions Processing Systems*, Morgan Kaufman, 1991.
76. M. Ubell, "The intelligent database machine," in *Query Processing in DBMS*, Springer-Verlag, 1985.
77. J. Ullman, "Implementation of logic query languages for databases," *ACM Trans. Database Systems*, vol. 10, no. 3, 1985.
78. P. Valduriez (ed.), *Data Management and Parallel Processing*, Chapman and Hall, London, 1992.
79. P. Valduriez: "Join indices," *ACM Trans. Database Systems*, vol. 12, no. 2, 1987.
80. P. Valduriez and G. Gardarin,, "Join and semi-join algorithms for a multiprocessor database machine," *ACM Trans. Database Systems*, vol. 9, no. 1, 1984.
81. P. Valduriez and G. Gardarin, *Analysis and Comparison of Relational Database Systems*, Addison-Wesley, Reading, MA, 1990.
82. P. Valduriez and S. Khoshafian, "Parallel evaluation of the transitive closure of a database relation," *Int. J. Parallel Programming*, vol. 12, no. 1, 1988.
83. P. Valduriez et al., "Compiling FAD, a database programming language," in *Int. Workshop Database Programming Languages*, Portland, Oregon, 1989.
84. G. Weikum, P. Zabback, and P. Scheuermann, "Dynamic file allocation in disk arrays," in *ACM-SIGMOD Int. Conf.*, Denver, 1991.
85. J. Wolf, D. Dias, P. Yu, and J. Turek, "An effective algorithm for parallelizing hash joins in the presence of data skews," in *Int. Conf. Data Engineering*, Kobe, Japan, 1991.
86. S. Zdonik and D. Maier (eds.), *Reading in Object-Oriented Database Systems*, Morgan Kaufmann, 1990.