DISTRIBUTED DATABASE SYSTEMS: WHERE ARE WE NOW?

M. Tamer Özsu[†]
GTE Laboratories Incorporated
40 Sylvan Road
Waltham, MA 02254
mto@gte.com

Patrick Valduriez
INRIA, Rocquencourt
78153 Le Chesnay
France
patrickv@madonna.inria.fr

ABSTRACT

Distributed database technology is expected to have a significant impact on data processing in the upcoming years. With the introduction of commercial products, expectations are that distributed database management systems will by and large replace centralized ones within the next decade. In this paper, we reflect on the promises of distributed database technology, take stock of where we are, and discuss the issues that remain to be solved. We also highlight new research issues that arise with the introduction of new technology and the subsequent relaxation of some of the assumptions underlying current systems.

Keywords: distributed database, data distribution, transaction management, distributed query processing, object-oriented system, knowledge base, multidatabase system.

Appeared in *IEEE Computer*, Vol. 24, No. 8, August 1991.

[†] On leave from Department of Computing Science, University of Alberta, Edmonton, Canada T6G 2H1; email: ozsu@cs.ualberta.ca.

1. INTRODUCTION

Distributed database technology is one of the more important developments of the past decade. During this period, distributed database research issues have been topics of intense study, culminating in the release of a number of “first generation” commercial products. Distributed database technology is expected to impact data processing the same way that centralized systems did a decade ago. It has been claimed that within the next ten years, centralized database managers will be an “antique curiosity” and most organizations will move toward distributed database managers [1, page 189].

The technology is now at the critical stage of finding its way into commercial products. At this juncture, it is important to seek answers to the following questions:

1. What were the initial goals and promises of the distributed database technology? How do the current commercial products measure up to these promises? In retrospect, were these goals achievable?

2. Have the important technical problems already been solved?

3. What are the technological changes that underlie distributed data managers and how will they impact next generation systems?

The last two questions hold particular importance for researchers since their answers lay down the road map for research in the upcoming years. Recent papers that address these questions have emphasized scaling problems [2] and issues related to the introduction of heterogeneity and autonomy [3]. While these problems are important ones to address, there are many others that remain unsolved. Even the much studied topics such as distributed query processing and transaction management have research problems that have yet to be addressed adequately. Furthermore, new issues arise with the changing technology, expanding application areas, and the experience that has been gained with the limited application of the distributed database technology.

In this paper our purpose is to address the above questions. Our emphasis is on answering these questions rather than providing a tutorial introduction to distributed database technology or a survey of the capabilities of existing products.

2. WHAT IS A DISTRIBUTED DATABASE SYSTEM?

A distributed database (DDB) is a collection of multiple, logically interrelated databases distributed over a computer network [4]. A distributed database management system (distributed DBMS) is then defined as the software system that permits the management of the DDB and makes the distribution transparent to the users. We use the term distributed database system (DDBS) to refer to the combination of the DDB and the distributed DBMS. Assumptions regarding the system that underlie these definitions are:

1. Data is stored at a number of sites. Each site is assumed to *logically* consist of a single processor. Even if some sites are multiprocessor machines, the distributed DBMS is not concerned with the storage and management of data on this parallel machine.

2. The processors at these sites are interconnected by a computer network rather than a multiprocessor configuration. The important point here is the emphasis on loose-interconnection between processors which have their own operating systems and operate independently. Even though shared-nothing multiprocessor architectures are quite similar to the loosely

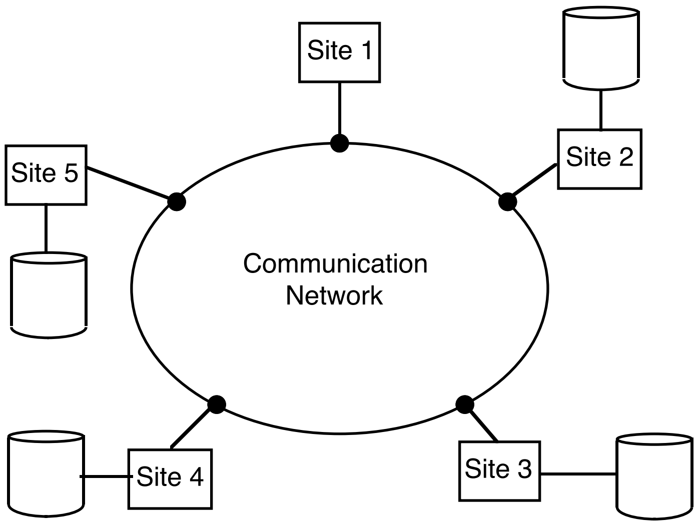

Figure 1. A Distributed Database Environment

interconnected distributed systems, they have different issues to deal with (e.g., task allocation and migration, load balancing, etc) that are not considered in this paper.

3. The DDB is a database, not some "collection" of files that can be individually stored at each node of a computer network. This is a distinction between a DDB and a collection of files managed by a distributed file system. To form a DDB, distributed data should be logically related, where the relationship is defined according to some structural formalism, and access to data should be at a high level via a common interface. The typical formalism that is used for establishing the logical relationship is the relational model. In fact, most existing distributed database system research assumes a relational system.

4. The system has the full functionality of a DBMS. It is neither, as indicated above, a distributed file system, nor a transaction processing system. Transaction processing is not only one type of distributed application, but it is also among the functions provided by a distributed DBMS. However, a distributed DBMS provides other functions such as query processing, structured organization of data, and so on that transaction processing systems do not necessarily deal with.

These assumptions are valid in today's technology base. Most of the existing distributed systems are built on top of local area networks in which each site is usually a single computer. The database is distributed across these sites such that each site typically manages a single local database (Figure 1). However, next generation distributed DBMSs will be designed differently due to the effects of technological developments – especially the emergence of affordable multiprocessors and high-speed networks – the increasing use of database technology in application domains which are more complex than business data processing, and the wider adoption of client-server mode of computing accompanied by the standardization of the interface between the clients and the servers. Thus, the next generation distributed DBMS environment will include multiprocessor database servers connected to high speed networks which link them and other data repositories to client machines that run application code and participate in the execution of database requests. Distributed relational DBMSs of this type are already appearing and a number of the existing object-oriented systems also fit this description.

A distributed DBMS as defined above is only one way of providing database management support for a distributed computing environment. In [4] we present a working classification of possible design alternatives along three dimensions: autonomy, distribution, and heterogeneity.

- *Autonomy* refers to the distribution of control, and indicates the degree to which individual DBMSs can operate independently. It involves a number of factors such as whether the component systems exchange information[1], whether they can independently execute transactions, and whether one is allowed to modify them. Three types of autonomy are tight integration, semiautonomy and full autonomy (or total isolation). In tightly integrated systems a single-image of the entire database is available to users who want to share the information which may reside in multiple databases. Semiautonomous systems consist of DBMSs that can (and usually do) operate independently, but have decided to participate in a federation to make their local data shareable. In totally isolated systems, however, the individual components are stand-alone DBMSs which know neither of the existence of other DBMSs nor of how to communicate with them.

- *Distribution* dimension of the taxonomy deals with data. We consider two cases, namely, either data are physically distributed over multiple sites that communicate with each other over some form of communication medium or they are stored at only one site.

- *Heterogeneity* can occur in various forms in distributed systems, ranging from hardware heterogeneity and differences in networking protocols to variations in data managers. The important ones from the perspective of database systems relate to data models, query languages, interfaces, and transaction management protocols. The taxonomy classifies DBMSs as homogeneous or heterogeneous.

The alternative system architectures based on this taxonomy are illustrated in Figure 2. The arrows along the axes do not indicate an infinite number of choices, but simply the dimensions of the taxonomy that we discussed above. For most of this paper, we deal with tightly integrated, distributed, and homogeneous database systems.

3. THE CURRENT STATE OF DISTRIBUTED DATABASE TECHNOLOGY

As with any emerging technology, DDBSs have their share of fulfilled and unfulfilled promises. In this section, we consider the commonly cited advantages of distributed DBMSs and discuss how well the existing commercial products provide these advantages.

3.1 Transparent Management of Distributed and Replicated Data

Centralized database systems have taken us from a paradigm of data processing, in which data definition and maintenance was embedded in each application, to one in which these functions are abstracted out of the applications and placed under the control of a server called the DBMS. This new orientation results in *data independence,* whereby the application programs are immune to changes in the logical or physical organization of the data and vice versa. The distributed database technology intends to extend the concept of data independence to environments where data is distributed and replicated over a number of machines connected by a network. This is provided by several forms of *transparency*: *network* (and, therefore, *distribution*) transparency, *replication* transparency, and *fragmentation* transparency. Transparent access to data separates the higher-level semantics of a system from lower-level implementation issues. Thus the database users would see a logically integrated, single image database even though it may be physically

[1]In this context "exchanging information" does not refer to networking concerns, but whether the DBMSs are designed to exchange information and coordinate their actions in executing user requests.

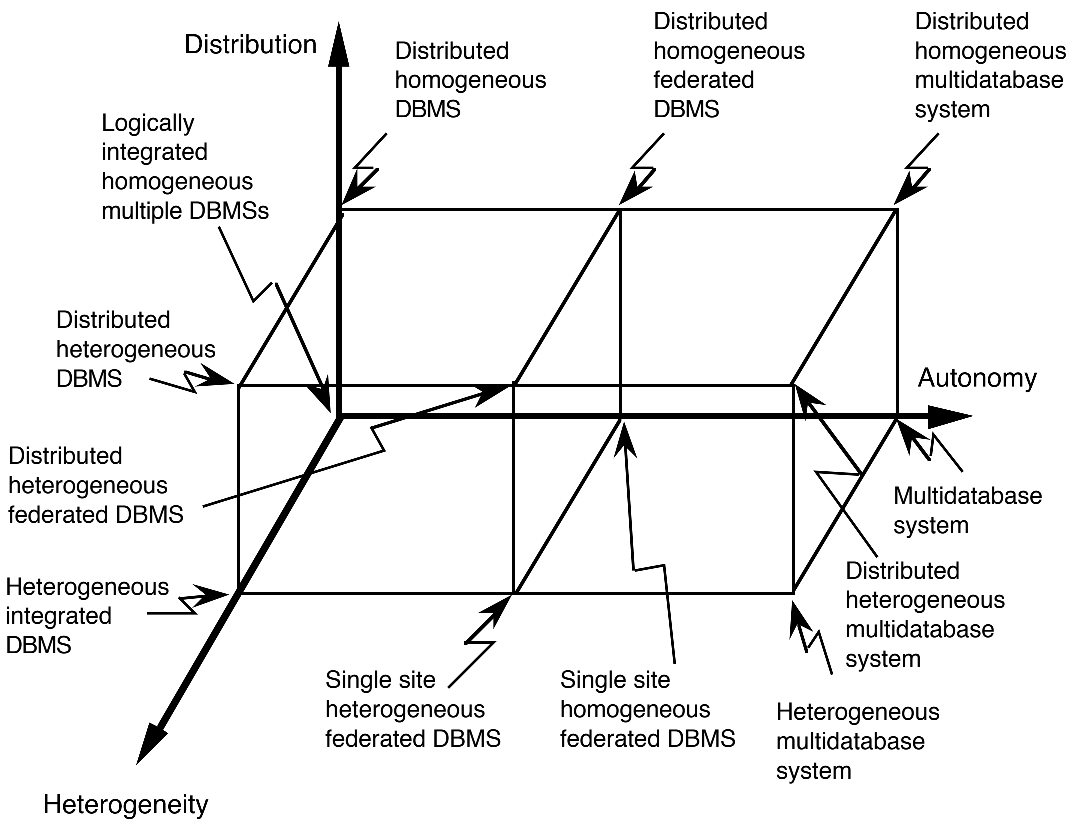
Distribution
Distributed homogeneous DBMS
Distributed homogeneous federated DBMS
Distributed homogeneous multidatabase system
Logically integrated homogeneous multiple DBMSs
Distributed heterogeneous DBMS
Autonomy
Distributed heterogeneous federated DBMS
Multidatabase system
Distributed heterogeneous multidatabase system
Heterogeneous integrated DBMS
Single site heterogeneous federated DBMS
Single site homogeneous federated DBMS
Heterogeneous multidatabase system
Heterogeneity

Figure 2. Implementation Alternatives

distributed, enabling them to access the distributed database as if it was a centralized one. In its ideal form, full transparency would imply a query language interface to the distributed DBMS which is no different from that of a centralized DBMS.

Most commercial distributed DBMSs do not provide a sufficient level of transparency. Part of this is due to the lack of support for the management of replicated data. A number of systems do not permit replication of the data across multiple databases while those that do require that the user be physically “logged on” to one database at a given time. Some distributed DBMSs attempt to establish their own transparent naming scheme, usually with unsatisfactory results, requiring the users either to specify the full path to data or to build aliases to avoid long path names. An important aspect of the problem is the lack of proper operating system support for transparency. Network transparency can easily be supported by means of a transparent naming mechanism by the operating system. The operating system can also assist with replication transparency, leaving the task of fragmentation transparency to the distributed DBMS.

Full transparency is not a universally accepted objective. Gray argues that full transparency makes the management of distributed data very difficult and claims that “applications coded with transparent access to geographically distributed databases have: poor manageability, poor modularity, and poor message performance” [5]. He proposes a remote procedure call mechanism between the requestor users and the server DBMSs whereby the users would direct their queries to a specific DBMS. We agree that the management of distributed data is more difficult if transparent access is provided to users, and that the client-server architecture with a remote procedure call-based communication between the clients and the servers is the right architectural approach. In fact, some commercial distributed DBMSs are organized in this fashion (e.g., Sybase). However, the original goal of distributed DBMSs to provide transparent access to distributed and replicated data should not be given up due to these difficulties. The issue is who should be taking over the responsibility of managing distributed and replicated data: the distributed DBMS or the user application? In our opinion, it should be the distributed DBMS whose components may be organized in a client-server fashion. The related technical issues are among the remaining research issues that need to be addressed.

3.2 Reliability Through Distributed Transactions

Distributed DBMSs are intended to improve reliability since they have replicated components and, thereby eliminate single points of failure. The failure of a single site, or the failure of a communication link which makes one or more sites unreachable, is not sufficient to bring down the entire system[2]. In the case of a distributed database, this means that some of the data may be unreachable, but with proper care, users may be permitted to access other parts of the distributed database. This “proper care” comes in the form of support for *distributed transactions*.

A *transaction* consists of a sequence of database operations, executed as an atomic action that transforms a consistent database state to another consistent database state even when a number of such transactions are executed concurrently (sometimes called *concurrency transparency*), and even when failures occur (also called *failure atomicity*). Therefore, a DBMS that provides full transaction support guarantees that concurrent execution of user transactions will not violate database consistency in the face of system failures as long as each transaction is correct, i.e., obeys the integrity rules specified on the database.

Distributed transactions execute at a number of sites at which they access the local database. With full support for distributed transactions, user applications can access a single logical image of the database and rely on the distributed DBMS to ensure that their requests will be executed correctly no matter what happens in the system. “Correctly” means that user applications do not need to be concerned with coordinating their accesses to individual local databases nor do they need to worry about the possibility of site or communication link failures during the execution of their transactions. This illustrates the link between distributed transactions and transparency, since both involve issues related to distributed naming and directory management, among other things.

Providing transaction support requires the implementation of distributed concurrency control and distributed reliability protocols, which are significantly more complicated than their centralized counterparts. The typical distributed concurrency control algorithm is some variation of the well-known two-phase locking (2PL) depending upon the placement of the lock tables and the assignment of the lock management responsibilities. Distributed reliability protocols consist of distributed commit protocols and recovery procedures. Commit protocols enforce atomicity of distributed transactions by ensuring that a given transaction has the same effect (commit or abort) at each site where it exists, whereas the recovery protocols specify how the global database consistency is to be restored following failures. In the distributed environment, the commit protocols are two-phase (2PC). In the first phase, an agreement is established among the various sites regarding the fate of a transaction. The agreed upon action is taken in the second phase.

Data replication increases database availability since copies of the data stored at a failed or unreachable site (due to link failure) exist at other operational sites. However, supporting replicas re-

[2]We do not wish to discuss the differences between site failures and link failures at this point. It is well-known that link failures may cause network partitioning and are, therefore, more difficult to deal with.

quire the implementation of *replica control protocols* that enforce a specified semantics of accessing them. The most straightforward semantics is *one-copy equivalence* which can be enforced by the ROWA protocol (“read one write all”). In ROWA, a logical read operation on a replicated data item is converted to one physical read operation on any one of its copies, but a logical write operation is translated to physical writes on all of the copies. More complicated replica control protocols that are less restrictive and that are based on deferring the writes on some copies have been studied, but are not implemented in any of the systems that we know.

Concurrency control and commit protocols are among the two most studied topics in distributed database research. Yet, their implementation in existing commercial systems is not widespread. The performance implications of implementing distributed transactions, which are not fully understood, make them unpopular among vendors. Commercial systems provide varying degrees of distributed transaction support. Some (e.g., Oracle) require users to have one database open at a given time, thereby eliminating the need for distributed transactions while others (e.g., Sybase) implement the basic primitives that are necessary for the 2PC protocol, but require the user applications to handle the coordination of the commit actions. In other words, the distributed DBMS does not enforce atomicity of distributed transactions, but provide the basic primitives by which user applications can enforce it. There are other systems, however, that implement the 2PC protocols fully (e.g., Ingres and NonStop SQL).

3.3 Improved Performance

The case for the improved performance of distributed DBMSs is typically made based on two points. First, a distributed DBMS fragments the conceptual database, enabling data to be stored in close proximity to its points of use (also called *data localization*). This has two potential advantages: (1) since each site handles only a portion of the database, contention for CPU and I/O services is not as severe as for centralized databases, and (2) localization reduces remote access delays that are usually involved in wide area networks (for example, the minimum round-trip message propagation delay in satellite-based systems is about 1 second). Most distributed DBMSs are structured to gain maximum benefit from data localization. Full benefits of reduced contention and reduced communication overhead can be obtained only by a proper fragmentation and distribution of the database.

Second, the inherent parallelism of distributed systems may be exploited for inter-query and intra-query parallelism. *Inter-query parallelism* results from the ability to execute multiple queries at the same time while *intra-query parallelism* is achieved by breaking up a single query into a number of subqueries each of which is executed at a different site, accessing a different part of the distributed database.

If the user access to the distributed database consisted only of querying (i.e., read-only access), then provision of inter-query and intra-query parallelism would imply that as much of the database as possible should be replicated. However, since most database accesses are not read-only, the mixing of read and update operations requires the implementation of elaborate concurrency control and commit protocols.

Existing commercial systems employ two alternative execution models (other than the implementation of full distributed transaction support) in realizing improved performance. The first alternative is to have the database open only for queries (i.e., read-only access) during the regular operating hours while the updates are batched. The database is then closed to query activity during off-hours when the batched updates are run sequentially. This is time multiplexing between read activity and update activity. A second alternative is based on multiplexing the database. Accordingly, two copies of the database are maintained, one for ad hoc querying (called the *query database*) and the other for updates by application programs (called the *production database*). At regular intervals, the production database is copied to the query database. This second alternative does not eliminate the need to implement concurrency control and reliability protocols for the production database since these are necessary to synchronize the write operations on the same data; however, it improves the performance of the queries since they can be executed without the overhead of transaction manipulation.

The performance characteristics of distributed database systems are not very well understood. There are not a sufficient number of true distributed database applications to provide a sound base to make practical judgements. In addition, the performance models of distributed database systems are not sufficiently developed. The database community has developed a number of benchmarks to test the performance of transaction processing applications, but it is not clear whether they can be used to measure the performance of distributed transaction management. The performance of the commercial DBMS products, even with respect to these benchmarks, are generally not openly published. NonStop SQL is one product for which performance figures, as well as the experimental setup that is used in obtaining them, has been published.

3.4 Easier and More Economical System Expansion

In a distributed environment, it should be easier to accommodate increasing database sizes. Major system overhauls are seldom necessary; expansion can usually be handled by adding processing and storage power to the system. We may call this *database size scaling*, as opposed to *network scaling* discussed later. It may not be possible to obtain a linear increase in “power,” since this also depends on the overhead of distribution. However, significant improvements are still possible.

Microprocessor and workstation technologies have played a role in improving economies. The price/performance characteristics of these systems make it more economical to put together a system of smaller computers with the equivalent power of a single big machine. Many commercial distributed DBMSs operate on minicomputers and workstations to take advantage of their favorable price/performance characteristics. The current reliance on workstation technology is because most of the commercial distributed DBMSs operate within local area networks for which the workstation technology is most suitable. The emergence of distributed DBMSs that run on wide-area networks may increase the importance of mainframes. On the other hand, future distributed DBMSs may support hierarchical organizations where sites consist of clusters of computers communicating over a local area network with a high-speed backbone wide area network connecting the clusters.

Another economic factor is the trade-off between data communication and telecommunication costs. In the previous section, we argued that data localization improves performance by reducing delays. It also reduces costs. Consider an application (such as inventory control) that needs to run at a number of locations. If this application accesses the database frequently, it may be more economical to distribute the data and to process it locally. This is in contrast to the execution of the application at various sites and making remote accesses to a central database that is stored at another site. In other words, the cost of distributing data and shipping some of it from one site to the other from time to time to execute distributed queries may be lower than the telecommunication cost of frequently accessing a remote database. We should state that this part of the economics argument is still speculative. As we indicated above, most of the distributed DBMSs are local area network products, and how they can be extended to operate in wide area networks is a topic of discussion and controversy.

4. UNSOLVED PROBLEMS

In the previous section we discussed the current state of commercial distributed DBMSs and how well they meet the original objectives that were set for the technology. Obviously, there is still some way to go before the commercial state-of-the-art fulfills the original goals of the technology. The issue is not only that the commercial systems have to catch up and implement the research results, but that there are still significant research problems that remain to be solved. The purpose of this section is to discuss these issues in some detail.

4.1 Network Scaling Problems

As noted before, the database community does not have a full understanding of the performance implications of all the design alternatives that accompany the development of distributed DBMSs. Specifically, questions have been raised about the scalability of some protocols and algorithms as the systems become geographically distributed [2] or as the number of system components increase [3]. Of specific concern is the suitability of the distributed transaction processing mechanisms (i.e., the two-phase locking, and, particularly, two-phase commit protocols) in wide area network-based distributed database systems. As mentioned before, there is a significant overhead associated with these protocols and implementing them over a slow wide area network may face difficulties [2].

Scaling issues are only one part of a more general problem, namely that we don't have a good handle on the role of the network architectures and protocols in the performance of distributed DBMSs. Almost all the performance studies that we know assume a very simple network cost model, sometimes as unrealistic as using a fixed communication delay that is independent of all network characteristics such as the load, message size, network size and so on. The inappropriateness of these models can be demonstrated easily. Consider, for example, a distributed DBMS that runs on an Ethernet-type local area network. Message delays in Ethernet increase as the network load increases, and, in general, cannot be bounded. Therefore, realistic performance models of an Ethernet-based distributed DBMS cannot realistically use a constant network delay or even a delay function which does not consider network load. In general, the performance of the proposed algorithm and protocols in different local area network architectures is not well understood, let alone their comparative behavior in moving from local area neetworks to wide area networks.

The proper way to deal with scalability issues is to develop general and sufficiently powerful performance models, measurement tools and methodologies. Such work for centralized DBMSs have been going on for some time, but have not yet been sufficiently extended to distributed DBMSs. We already raised questions about the suitability of the existing benchmarks. Detailed and comprehensive simulation studies do not exist either. Even though there are plenty of performance studies of distributed DBMS, these usually employ simplistic models, artificial workloads, conflicting assumptions or consider only a few special algorithms. It has been suggested that to make generalizations based on the existing performance studies requires a giant leap of faith. This does not mean that we do not have some understanding of the trade-offs. In fact, certain trade-offs have long been recognized and even the earlier systems have considered them in their design. For example, the query processor of the SDD-1 system was designed to execute distributed operations most efficiently on slow wide area networks. Later studies considered the optimization of query processors in faster, broadcasting local area networks. However, these trade-offs can mostly be spelled out only in qualitative terms; their quantification requires more research on performance models.

4.2 Distribution Design

The design methodology of distributed databases varies according to the system architecture. In the case of tightly integrated distributed databases, design proceeds top-down from requirements analysis and logical design of the *global database* to physical design of each *local database*. In the case of distributed multidatabase systems, the design process is bottom-up and involves the integration of existing databases. In this section we concentrate on the top-down design process issues.

The step in the top-down process that is of interest to us is *distribution design*. This step deals with designing the *local conceptual schemas* by distributing the global entities over the sites of the distributed system. The global entities are specified within the *global conceptual schema*. In case of the relational model, both the global and the local entities are relations and distribution design maps global relations to local ones. The most important research issue that requires attention is the development of a practical distribution design methodology and its integration into the general data modeling process.

There are two aspects of distribution design: *fragmentation* and *allocation*. Fragmentation deals with the partitioning of each global relation into a set of fragment relations while allocation concentrates on the (possibly replicated) distribution of these local relations across the sites of the distributed system. Research on fragmentation has concentrated on horizontal (i.e., selecting) and vertical (i.e., projecting) fragmentation of global relations. Numerous allocation algorithms based on mathematical optimization formulations have also been proposed.

There is no underlying design methodology that combines the horizontal and vertical partitioning techniques; horizontal and vertical partitioning algorithms have been developed completely independently. If one starts with a global relation, there are algorithms to decompose it horizontally as well as algorithms to decompose it vertically into a set of fragment relations. However, there are no algorithms that fragment a global relation into a set of fragment relations some of which are decomposed horizontally and others vertically. It is always pointed out that most real-life fragmentations would be mixed, i.e., would involve both horizontal and vertical partitioning of a relation, but the methodology research to accomplish this is lacking. What is needed is a distribution design methodology which encompasses the horizontal and vertical fragmentation algorithms and uses them as part of a more general strategy. Such a methodology should take a global relation together with a set of design criteria and come up with a set of fragments some of which are obtained via horizontal and others obtained via vertical fragmentation.

The second part of distribution design is allocation which is typically treated independently of fragmentation. The process is, therefore, linear, where the output of fragmentation is input to allocation. At first sight, the isolation of the fragmentation and the allocation steps appears to simplify the formulation of the problem by reducing the decision space. However, closer examination reveals that isolating the two steps actually contributes to the complexity of the allocation models. Both steps have similar inputs, differing only in that fragmentation works on global relations whereas allocation considers fragment relations. They both require information about the user applications (e.g., how often they access data, what the relationship of individual data objects to one another is, etc), but ignore how each other makes use of these inputs. The end result is that the fragmentation algorithms decide how to partition a relation based partially on how applications access it, but the allocation models ignore the part that this input plays in fragmentation. Therefore, the allocation models have to include all over again detailed specification of the relationship among the fragment relations and how user applications access them. What would be more promising is to extend the methodology discussed above so that the interdependence of the fragmentation and the allocation decisions is properly reflected. This requires extensions to existing distribution design strategies (e.g., [6]).

We recognize that integrated methodologies such as the one we propose here may be considerably complex. However, there may be synergistic effects of combining these two steps enabling the development of quite acceptable heuristic solution methods. There are some studies that give us hope that such integrated methodologies and proper solution mechanisms can be developed. These methodologies build a simulation model of the distributed DBMS, taking as input a specific database design, and measure its effectiveness. Development of tools based on such methodologies, which aid the human designer rather than attempt to replace him, is probably the more appropriate approach to the design problem.

4.3 Distributed Query Processing

Distributed query processors automatically translate a high-level query on a distributed database, which is seen as a single database by the users, into an efficient low-level execution plan expressed on the local databases. 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 the equivalence between relational calculus and relational algebra, and the transformation rules associated with relational algebra. Second, the execution plan must be "optimal," i.e., it must minimize a cost function that captures resource consumption. This requires investigating equivalent alternative plans in order to select the best one.

Because of the difficulty of addressing these two aspects together, they are typically isolated in two sequential steps which we call *data localization* and *global optimization* in [4]. These steps are generally preceded by query decomposition which simplifies the input query and rewrites it in relational algebra. Data localization transforms an input algebraic query expressed on the distributed database into an equivalent fragment query (i.e., a query expressed on database fragments stored at different sites) which can be further simplified by algebraic transformations. Global query optimization generates an optimal execution plan for the input fragment query by making decisions regarding operation ordering, data movement between sites and the choice of both distributed and local algorithms for database operations. There are a number of problems regarding this last step. They have to do with the restrictions imposed on the cost model, the focus on a subset of the query language, the trade-off between optimization cost and execution cost, and the optimization/reoptimization interval.

The cost model is central to global query optimization since it provides the necessary abstraction of the distributed DBMS execution system in terms of access methods, and an abstraction of the database in terms of physical schema information and related statistics. The cost model is used to predict the execution cost of alternative execution plans for a query. A number of important restrictions are often associated with the cost model, limiting the effectiveness of optimization in improving throughput. Work in extensible query optimization [7] can be useful in parameterizing the cost model which can then be refined after much experimentation.

Even though query languages are becoming increasingly powerful (e.g., new versions of SQL), global query optimization typically focuses on a subset of the query language, namely select-project-join (SPJ) queries with conjunctive predicates. This is an important class of queries for which good optimization opportunities exist. As a result, a good deal of theory has been developed for join and semijoin ordering. However, there are other important queries that warrant optimization, such as queries with disjunctions, unions, fixpoint, aggregations or sorting. A promising solution is to separate the language understanding from the optimization itself which can be dedicated to several optimization "experts."

There is a necessary trade-off between *optimization cost* and *quality* of the generated execution plans. Higher optimization costs are probably acceptable to produce "better" plans for repetitive queries, since this would reduce query *execution cost* and amortize the optimization cost over many executions. However, it is unacceptable for ad hoc queries which are executed only once. The optimization cost is mainly incurred by searching the solution space for alternative execution plans. In a distributed system, the solution space can be quite large because of the wide range of distributed execution strategies. Therefore, it is critical to study the application of efficient search strategies that avoid the exhaustive search approach.

Global query optimization is typically performed prior to the execution of the query, hence called static. A major problem with this approach is that the cost model used for optimization may become inaccurate because of changes in the fragment sizes or database reorganization which is important for load balancing. The problem, therefore, is to determine the optimal intervals of re-compilation/reoptimization of the queries taking into account the trade-off between optimization and execution cost.

4.4 Distributed Transaction Processing

It may be hard to believe that in an area as widely researched as distributed transaction processing there may still be important topics to investigate, but there are. We have already discussed the scaling problems of transaction management algorithms. Additionally replica control protocols, more sophisticated transaction models, and non-serializable correctness criteria require further attention.

In replicated distributed DBMSs, database operations are specified on logical data objects[3]. The replica control protocols are responsible for mapping an operation on a logical data object to an operation on multiple physical copies of this data object. In so doing, they ensure the *mutual consistency* of the replicated database. The ROWA rule that we discussed earlier is the most straightforward method of enforcing mutual consistency. Accordingly, a replicated database is in a mutually consistent state if all the copies of every data object have identical values.

The field of data replication needs further experimentation, research on replication methods for computation and communication, and more work to enable the systematic exploitation of application-specific properties. Experimentation is required to evaluate the claims that are made by algorithm and system designers, and we lack a consistent framework for comparing competing techniques. One of the difficulties in quantitatively evaluating replication techniques lies in the absence of commonly accepted *failure incidence models*. For example, Markov models that are sometimes used to analyze the availability achieved by replication protocols assume the statistical independence of individual failure events, and the rarity of network partitions relative to site failures. We do not currently know that either of these assumptions is tenable, nor do we know how sensitive Markov models are to these assumptions. The validation of the Markov models by simulation cannot be trusted in the absence of empirical measurements, since simulations often embody the same assumptions that underlie the Markov analysis. Thus, there is a need for empirical studies to monitor failure patterns in real-life production systems, with the purpose of constructing a simple model of typical *failure loads*.

To achieve the twin goals of data replication, namely *availability* and *performance*, we need to provide integrated systems in which the replication of data goes hand in hand with the replication of computation and communication (including I/O). Only data replication has been studied intensely; relatively little has been done in the replication of computation and communication. Replication of computation has been studied for a variety of purposes, including running synchronous duplicate processes as "hot standbys," and processes implementing different versions of the same software to guard against human design errors. Replication of communication messages primarily by retry has been studied in the context of providing reliable message delivery, and a few papers report on the replication of input/output messages to enhance the availability of transactional systems. However, more work needs to be done to study how these tools may be integrated together with data replication to support such applications as real time control systems, that may benefit from all three kinds of replication. This work would be invaluable in guiding operating system

[3]We use the term "data object" here instead of the more common "data item" because we do not want to make a statement about the granularity of the logical data.

and programming language designers towards the proper set of tools to offer in support of fault-tolerant systems.

In addition to replication, but related to it, work is required on more elaborate transaction models, especially those that exploit the semantics of the application. Higher availability and performance, as well as concurrency, can be achieved this way. As database technology enters new application domains such as engineering design, software development and office information systems, the nature and requirements for transactions change. Thus, work is needed on more complicated transaction models and on correctness conditions different from serializability.

As a first approximation, the existing work on transaction models can be classified along two dimensions: the transaction model and the structure of objects that they operate on. Along the transaction model dimension, we recognize flat transactions, closed nested transactions and open nested transaction models such as sagas and the like, and models that include both open and closed nesting, in increasing order of complexity. Along the object structure dimension, we identify simple objects (e.g., pages), objects as instances of abstract data types (ADTs), and complex objects, again in increasing complexity. We make the distinction between the last two to indicate that objects as instances of abstract data types support encapsulation (and therefore are harder to run transactions on than simple objects), but do not have a complex structure (i.e., do not contain other objects) and their types do not participate in an inheritance hierarchy.

Within the above framework, most of the transaction model work in distributed systems has concentrated on the execution of flat transactions on simple objects. This point in the design space is quite well understood. While some work has been done in the application of nested transactions on simple objects, much remains to be done, especially in distributed databases. Specifically, the semantics of these transaction models are still being worked out. Similarly, there has been work done on applying simple transactions to objects as instances of abstract data types and to complex objects. Again, these are initial attempts which need to be followed up to specify their full semantics, their incorporation into a DBMS, their interaction with recovery managers and, finally, their distribution properties.

Complex transaction models are important in distributed systems for a number of reasons. First, transaction processing in distributed multidatabase systems can benefit from the relaxed semantics of these models. Second, the new application domains that distributed DBMSs will support in the future (e.g., engineering design, office information systems, cooperative work, etc) require transaction models that incorporate more abstract operations that execute on complex data. Furthermore, these applications have a different sharing paradigm than the typical database access that we are accustomed to. For example, computer-assisted cooperative work environments require participants to cooperate in accessing shared resources rather than competing for them as is usual in typical database applications. These changing requirements necessitate the development of new transaction models and accompanying correctness criteria.

4.5 Integration with Distributed Operating Systems

The undesirability of running a centralized or distributed DBMS as an ordinary user application on top of a host operating system (OS) has long been recognized [8, 9]. There is a mismatch between the requirements of the DBMS and the functionality of the existing OSs. This is even more true in the case of distributed DBMSs which require functions (e.g., distributed transaction support including concurrency control and recovery, efficient management of distributed persistent data, more complicated access methods) that existing distributed OSs do not provide. Furthermore, distributed DBMSs necessitate modifications in how the distributed OSs perform their traditional functions (e.g., task scheduling, naming, buffer management). In this section, we briefly highlight the fundamental issues in distributed DBMS/distributed OS integration. They relate to the system architecture, transparent naming of resources, persistent data management, distributed scheduling, remote communication and transaction support. A more detailed discussion of the issues can be found in Chapter 13 of [4].

An important architectural consideration is that the coupling of distributed DBMSs and distributed OSs is not a binary integration issue. There is also the communication network protocols that need to be considered, adding to the complexity of the problem. Thus the architectural paradigm has to be flexible enough to accommodate distributed DBMS functions, distributed operating system services as well as the communication protocol standards such as the ISO/OSI or IEEE 802. In this context, efforts that include too much of the database functionality inside the operating system kernel or those that modify tightly-closed operating systems are likely to prove unsuccessful. In our view, the operating system should only implement the essential OS services and those DBMS functions that it can *efficiently* implement and then *should get out of the way*. The model that best fits this requirement seems to be the client-server architecture with a small kernel that provides the database functionality that can efficiently be provided and does not hinder the DBMS in efficiently implementing other services at the user level (e.g., Mach, Amoeba). Object-orientation may also have a lot to offer as a system structuring approach to facilitate this integration.

Naming is the fundamental mechanism that is available to the operating system for providing transparent access to system resources. Whether or not access to distributed objects should be transparent at the operating system level is a contentious issue involving the tradeoff between flexibility of data management and ease of use on the one hand and system overhead on the other. From the perspective of a distributed DBMS, transparency is important. As we indicated earlier, many of the existing distributed DBMSs attempt to establish their own transparent naming schemes without significant success. More work is necessary in investigating the naming issue and the relationship between distributed directories and OS name servers. A worthwhile naming construct that deserves some attention in this context is the capability concept which was used in older systems such as Hydra and is being used in more modern OSs such as Amoeba.

Storage and management of *persistent data* which survive past the execution of the program that manipulates them is the primary function of database management systems. Operating systems have traditionally dealt with persistent data by means of file systems. If a successful cooperation paradigm can be found, it may be possible to use the DBMS as the OS file system. At a more general level, the cooperation between programming languages, DBMS and OS to manage persistent data requires further research. The distributed file systems do not address distributed DBMS concerns because either they do not provide for concurrent access to data, or the granularity of sharing is too large.

Two communication paradigms that have been widely studied in distributed operating systems are message passing[4] and RPC. The relative merits of each approach have long been debated, but the simple semantics of RPC (blocking, one time execution) have been appealing to distributed system designers. As discussed before, an RPC-based access to distributed data at the user level is sometimes proposed in place of providing fully transparent access [5]. However, implementation of an RPC mechanism for a heterogeneous computing environment is not an easy matter. The issue is that the RPC systems of different vendors do not interoperate. It may be necessary to look at communication at higher levels of abstraction in order to overcome heterogeneity or at lower levels of abstraction (i.e., message passing) to achieve more parallelism. This tradeoff needs to be further studied.

[4]Note that we are referring to logical message passing, not to physical. Remote procedure calls have to be transmitted between sites as physical messages as well.

In current DBMSs, the transaction manager is implemented as part of the DBMS. Whether transactions should and can be implemented as part of standard operating system services has long been discussed. It is fair to state that there are strong arguments on both sides, but a clear resolution of the issue requires more research as well as some more experience with the various general purpose (i.e., non-DBMS) transaction management services.

4.6 Distributed Multidatabase Systems

As we indicated, multidatabase system organization is an alternative to logically integrated distributed databases. The fundamental difference between the two approaches is the level of autonomy afforded to the component data managers at each site. While integrated DBMSs have components which are designed to work together, multidatabase management systems (multi-DBMS) consist of components which may not have any notion of cooperation. Specifically, the components are independent DBMSs, which means, for example, that while they may have facilities to execute transactions, they have no notion of executing distributed transactions that span multiple components. In this section, we briefly highlight the open problems that relate to query processing and transaction management. We also address the standardization issues and their role in interoperability. More detailed discussion can be found in [10], among others.

The autonomy and potential heterogeneity of component systems create problems in query processing and especially in query optimization. The fundamental problem is the difficulty of global optimization when local cost functions are not known and local cost values cannot be communicated to the multi-DBMS. It has been suggested that semantic optimization based only on qualitative information may be the best that one can do, but semantic query processing is not fully understood either. Potentially, it may be possible to develop hierarchical query optimizers which perform some amount of global query optimization and then let each local system perform further optimization on the localized subquery. This may not provide an "optimal" solution, but may still enable some amount of optimization. The emerging standards, which we will discuss shortly, may also make it easier to share some cost information.

Transaction processing in autonomous multidatabase systems is made more difficult by the autonomy of the underlying DBMSs. Since they are autonomous, they have their own transaction processing services (i.e., transaction manager, scheduler, recovery manager) and are capable of accepting local transactions and running them to completion. The multi-DBMS layer has its own transaction processing components in charge of accepting global transactions which access multiple databases and coordinating their execution. A global transaction is divided into subtransactions each of which is submitted to one of the component DBMSs. However, since the multi-DBMS is not aware of the local transactions, it cannot control the local conflicts, nor can it control *indirect conflicts* between global transactions caused by the interference of local transactions.

A number of different solutions have been proposed to deal with concurrent multidatabase transaction processing. Some of these use global serializability of transactions as their correctness criteria while others relax serializability. Most of this work should be treated as being preliminary initial attempts at understanding and formalizing the issues. There are many issues that remain to be investigated. One area of investigation has to deal with revisions in the transaction model and the correctness criteria. There are initial attempts to recast the transaction model assumptions and this work needs to continue. Nested transaction models look particularly promising for multidatabase systems and its semantics based on knowledge about the transaction's behavior needs to be formalized. In this context, it is necessary to go back and reconsider the meaning of consistency in multidatabase systems. A good starting point is the four degrees of consistency defined by Gray [8].

Another difficult issue that requires further investigation is the reliability and recovery aspects of multidatabase systems. The autonomy of individual DBMSs makes it difficult to incorporate 2PC into global transaction processing, which, in turn, makes it difficult to enforce distributed transaction atomicity. Even though the topic has been addressed in some recent studies, these approaches are initial engineering solutions. The development of reliability and recovery protocols for multidatabase systems and their integration with concurrency control mechanisms still needs to be worked out.

Probably one the fundamental impediments to further development of multidatabase systems is the lack of understanding of the nature of autonomy, which is a major contributor to the additional complexity of these systems. It is probably the case that what we call autonomy is itself composed of a number of factors. Thus, the nature of autonomy needs to be clearly and precisely characterized. Furthermore, most researchers treat autonomy as if it were a “all-or-nothing” feature. Even the taxonomy that we considered indicated only three points along this dimension. However, the spectrum between “no autonomy” and “full autonomy” probably contains many distinct points. It is essential, in our opinion, to (a) precisely define what is meant by “no autonomy” and “full autonomy”, (b) precisely delineate and define the many different levels of autonomy, and (c) identify, as we indicated above, the appropriate degree of database consistency that is possible for each of these levels. At that point, it would be more appropriate to discuss the different transaction models and execution semantics that are appropriate at each of these levels of autonomy. In addition, this process should enable the identification of a layered structure, similar to the ISO Open System Interconnection model, for the interoperability of autonomous and possibly heterogeneous database systems. Such a development would then make it possible to specify interfacing standards at different levels. Some standardization work is already under way within the context of the Remote Data Access (RDA) standard, and this line of work will make the development of practical solutions to the interoperability problem possible.

5. CHANGING TECHNOLOGY AND NEW ISSUES

Distribution is commonly identified as one of the major features of next generation database systems which will be ushered in by the penetration of database technology into new application areas with different requirements than traditional business data processing and the technological developments in computer architecture and networking. One of the common features of next-generation database systems is that the data model to be supported will need to be more powerful than the relational model, yet without compromising its advantages (data independence and high-level query languages). When applied to more complex application domains such as CAD/CAM, software design, office information systems, and expert systems the relational data model exhibits limitations in terms of complex object support, type system and rule management. To address these issues, two important technologies, knowledge bases and object-oriented databases, are currently being investigated. Another major issue is going to be system performance as more functionality is added. Exploiting the parallelism available in multiprocessor computers is one promising approach to provide high performance. Techniques designed for distributed databases can be useful but need to be significantly extended to implement parallel database systems.

Object-oriented databases (OODBs) [11] combine object-oriented programming and database technologies in order to provide higher modeling power and flexibility to data intensive application programmers. Over the last five years, OODBs have been the subject of intensive research and experimentation which led to an impressive number of prototypes and commercial products. However, the theory and practice of developing distributed object-oriented DBMSs (OODBMS) have yet to be fully developed. Dealing with distributed environments will make the problems even more difficult. Additionally, the issues related to data dictionary management and distributed object management have to be dealt with. However, distribution is an essential requirement, since applications which require OODB technology typically arise in networked workstation environments. The early commercial OODBMSs (e.g., GEMSTONE) use a client/server architecture where multiple workstations can access the database centralized on a server. However, distributing an OODB within a network of workstations (and servers) becomes very attractive. In fact, some OODBMSs already support some form of data distribution transparency (e.g., ONTOS and Distributed ORION).

Knowledge base management systems attempt to make database management more intelligent by managing knowledge in addition to data. Capturing knowledge in the form of rules has been extensively studied in a particular form of knowledge base system called a deductive database. Deductive database systems manage and process rules against large amounts of data within the DBMS rather than with a separate subsystem. Rules can be declarative (assertions) or imperative (triggers). Rule management 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, in particular, in the presence of recursive and negated predicates. However, much work is needed to combine rule support with object-oriented capabilities. For reasons similar to those for OODB applications, knowledge base applications are likely to arise in networked workstation environments. These applications can also arise in parallel computing environments when the database is managed by a multiprocessor database server (see next paragraph). In any case, there are a number of similar issues which can get simplified by relying on distributed relational database technology. Unlike most OODB approaches which try to extend an object-oriented programming language, this is a strong advantage for implementing knowledge bases in distributed environments. Therefore, the new issues have more to do with distributed knowledge management and processing and debugging of distributed knowledge base queries than with distributed data management.

Parallel database systems intend to exploit the recent multiprocessor computer architectures in order to build high-performance and fault-tolerant database servers [12]. This can be achieved by extending distributed database technology, for example, by fragmenting the database across multiple nodes so that more inter- and intra-query parallelism can be obtained. For obvious reasons such as set-oriented processing and application portability, most of the work in this area has focused on supporting SQL. There is a continuing discussion as to which of the shared-memory or distributed-memory multiprocessor architectures is more appropriate for data management. The resolution of this question will require more experimental study. The design problems of parallel database systems, such as the operating system support, data placement, parallel algorithms, and parallelizing compilation, are common to both kinds of multiprocessor architectures. If parallel data servers become prevalent, it is not difficult to see an environment where multiple of them are placed on a backbone network. This gives rise to distributed systems consisting of clusters of processors [5]. An interesting concern in such an environment is internetworking. Specifically, the execution of database commands which span multiple, and possibly heterogeneous, clusters creates at least the problems that we discussed under 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.

6. CONCLUSIONS

In this paper we discuss the state-of-the-art in distributed database research and products. Specifically, we (a) reviewed the initial goals and promises of the distributed database technology and commented on how well the current commercial products fulfill these promises; (b) discussed the degree to which the important technical problems have been addressed; and (c) briefly considered the technological changes that underlie distributed data managers to determine how they are likely to impact next generation systems.

The initial promises of distributed database systems, namely transparent management of distributed and replicated data, improved system reliability via distributed transactions, improved system performance by means of inter- and intra-query parallelism, and easier and more economical system expansion, are met to varying degrees by existing commercial products. Full realization of these promises is not only dependent upon the commercial state-of-the-art catching up with the research results, but is also dependent upon the solution of a number of problems. The issues that have been studied, but still require more work are the following:

1. performance models, methodologies and benchmarks to better understand the sensitivity of the proposed algorithms and mechanisms to the underlying technology;

2. distributed query processing to handle queries which are more complex than select-project-join, ability to process multiple queries at once to save on common work, and optimization cost models to be able to determine when such multiple query processing is beneficial;

3. advanced transaction models that differ from those that are defined for business data processing and that better reflect the mode of processing that is common in most distributed applications (i.e., cooperative sharing vs competition, interaction with the user, long duration) for which the distributed database technology is going to provide support;

4. analysis of replication and its impact on distributed database system architecture and algorithms and the development of efficient replica control protocols that improve system availability;

5. implementation strategies for distributed DBMSs that emphasize a better interface and cooperation with distributed operating systems;

6. theoretically complete and correct and practical design methodologies for distributed databases; and

7. full set of problems related to the interconnection of autonomous information processing systems.

In addition to these problems, the changing nature of the technology on which distributed DBMSs are implemented will make parallel database servers feasible. This will effect DDBSs in two ways. First, distributed DBMSs will be implemented on these parallel database servers, requiring the revision of most of the existing algorithms and protocols to operate on the parallel machines. Second, the parallel database servers will be connected as servers to networks, requiring the development of distributed DBMSs that will have to deal with a hierarchy of data managers. Furthermore, as distributed database technology infiltrates non-business data processing type application domains (e.g., engineering databases, office information systems, software development environments), the capabilities required of these systems will change. This will necessitate a shift in emphasis from relational systems to data models which are more powerful. Current research along these lines concentrates on object-orientation and knowledge base systems.

As this paper clearly demonstrates, there are many important technical problems that await solution, as well as new ones that arise as a result of the technological changes that underlie distributed data managers. These problems should keep researchers as well as distributed DBMS implementers quite busy for some time to come.

ACKNOWLEDGMENTS

We would like to extend our thanks to Abdelsalam Heddaya of Boston University who not only reviewed the entire paper and provided many comments, but also helped us with the replication part, which is based on a draft written by him. Alex Biliris of Boston University, Michael Brodie, Alex Buchmann, Dimitrios Georgakopoulos, and Frank Manola of GTE Laboratories have also read the entire manuscript and provided many suggestions regarding the content and the presentation that improved the paper significantly. The research of the first author has been partially supported by the Natural Sciences and Engineering Research Council (NSERC) of Canada under the operating grant OGP-0951.

REFERENCES

[1] M. STONEBRAKER. *Readings in Database Systems*. San Mateo, Calif.: Morgan Kaufmann, 1988.

[2] M. STONEBRAKER. "Future Trends in Database Systems," *IEEE Trans. Knowledge and Data Eng.*, Vol. 1, No. 1, March 1989, pp. 33–44.

[3] H. GARCIA-MOLINA AND B. LINDSAY. "Research Directions for Distributed Databases," *IEEE Q. Bull. Database Eng.*, Vol. 13, No. 4, December 1990, pp. 12–17.

[4] M.T. ÖZSU AND P. VALDURIEZ. *Principles of Distributed Database Systems*. Englewood Cliffs, NJ: Prentice-Hall, 1991.

[5] J. GRAY. *Transparency in its Place – The Case Against Transparent Access to Geographically Distributed Data*, Technical Report TR89.1, Tandem Computers Inc., Cupertino, CA, 1989.

[6] S. CERI, B. PERNICI, AND G. WIEDERHOLD. "Distributed Database Design Methodologies," *Proc. IEEE*, Vol. 75, No. 5, May 1987, pp. 533–546.

[7] C. FREYTAG. "A Rule-based View of Query Optimization," In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, San Francisco, 1987, pp. 173– 180.

[8] J. GRAY. "Notes on Data Base Operating Systems," In *Operating Systems: An Advanced Course*, R. Bayer, R. M. Graham, and G. Seegmüller (eds.), New York: Springer-Verlag, 1979, pp. 393–481.

[9] M. STONEBRAKER. "Operating System Support for Database Management," *Commun. ACM*, Vol. 24, No. 7, July 1981, pp. 412–418.

[10] A.K. ELMAGARMID AND C. PU (eds). *ACM Comp. Surv. Special Issue on Heterogeneous Databases*, Vol. 22, No. 3, September 1990.

[11] S. ZDONIK AND D. MAIER (eds.). *Readings in Object-Oriented Database Systems*, San Mateo, CA: Morgan Kaufmann, 1990.

[12] P. VALDURIEZ (ed.). *Data Management and Parallel Processing*, London, UK: Chapman and Hall, 1991 (to appear).