# The State of the Art in Distributed Query Processing

DONALD KOSSMANN

*University of Passau*

Distributed data processing is becoming a reality. Businesses want to do it for many reasons, and they often must do it in order to stay competitive. While much of the infrastructure for distributed data processing is already there (e.g., modern network technology), a number of issues make distributed data processing still a complex undertaking: (1) distributed systems can become very large, involving thousands of heterogeneous sites including PCs and mainframe server machines; (2) the state of a distributed system changes rapidly because the load of sites varies over time and new sites are added to the system; (3) legacy systems need to be integrated—such legacy systems usually have not been designed for distributed data processing and now need to interact with other (modern) systems in a distributed environment.

This paper presents the state of the art of query processing for distributed database and information systems. The paper presents the "textbook" architecture for distributed query processing and a series of techniques that are particularly useful for distributed database systems. These techniques include special join techniques, techniques to exploit intraquery parallelism, techniques to reduce communication costs, and techniques to exploit caching and replication of data. Furthermore, the paper discusses different kinds of distributed systems such as client-server, middleware (multitier), and heterogeneous database systems, and shows how query processing works in these systems.

Categories and Subject Descriptors: E.5 [**Data:**]: Files; H.2.4 [**Database Management Systems:**]: Distributed Databases, Query Processing; H.2.5 [**Heterogeneous Databases:**]: Data Translation

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Query optimization, query execution, client-server databases, middleware, multitier architectures, database application systems, wrappers, replication, caching, economic models for query processing, dissemination-based information systems

## 1. INTRODUCTION

### 1.1 Background and Motivation

Researchers and practitioners have been interested in distributed database systems since the 1970s. At that time, the main focus was on supporting distributed data management for large corporations and organizations that kept their data at different offices or subsidiaries.

This work was partially supported by the German Research Council (DFG) under contract Ke 401/7-1.

Author's address: University of Passau, 94030 Passau, Germany; email: http://www.db.fmi.unipassau.de/~kossmann.

Although there was a clear need and many good ideas and prototypes (e.g., System R* [Williams et al. 1981], SDD-1 [Bernstein et al. 1981], and Distributed Ingres [Stonebraker 1985]), the early efforts in building distributed database systems were never commercially successful [Stonebraker 1994]. In some aspects, the early distributed database systems were ahead of their time. First, communication technology was not stable enough to ship megabytes of data as required for these systems. Second, large businesses somehow managed to survive without sophisticated distributed database technology by sending tapes, diskettes, or just paper to exchange data between their offices.

Today, the situation has changed dramatically. Distributed data processing is both feasible and needed. Almost all major database system vendors offer products to support distributed data processing (e.g., IBM, Informix, Microsoft, Oracle, Sybase), and large database application systems have a distributed architecture (e.g., business application systems such as Baan IV, Oracle Finance, Peoplesoft 7.5, and SAP R/3). Distributed data processing is feasible because of recent technological advances (e.g., hardware, software protocols, standards). Distributed data processing is needed because of changing business requirements, which have made distributed data processing cost-effective and in certain situations the only viable option. Specifically, businesses are beginning to rely on distributed rather than centralized databases for the following reasons:

1. **Cost and scalability.** Today, one thousand PC processors are cheaper and significantly more powerful than one big mainframe computer. So, it makes economic sense to replace a mainframe by a network of small, off-the-shelf processors. Furthermore, it is very difficult to "up-size" a mainframe computer if a company grows, while new PCs can be added to the network at any time in order to meet a company's new requirements. High availability can be achieved by mirroring (replicating) data.
2. **Integration of different software modules.** It has become clear that no single software package can meet all the requirements of a company. Companies must, therefore, install several different packages, each potentially with its own database, and the result is a distributed database system. Even single software packages offered by one vendor have a distributed, component-based architecture so that the vendor can market and offer upgrades for every component individually.
3. **Integration of legacy systems.** The integration of legacy systems is one particular example that demonstrates how some companies are forced to rely on distributed data processing in which their old legacy systems need to coexist with new modern systems.
4. **New applications.** There are a number of new emerging applications that rely heavily on distributed database technology; examples are workflow management, computer-supported collaborative work, tele-conferencing, and electronic commerce.
5. **Market forces.** Many companies are forced to reorganize their businesses and use state-of-the-art distributed information technology in order to remain competitive. As an example, people will probably not eat more Pizza because of the Internet, but a Pizza delivery service is definitely going to lose some of its market share if it does not allow people to order Pizza on the Web.

This list shows that there are many different reasons to rely on distributed architectures and correspondingly many different kinds of distributed systems exist. Sometimes it is only the software and not the hardware that is distributed. The purpose of this paper is to give a comprehensive overview of what query processing techniques are needed to implement any kind of distributed database and information system. It is assumed that users and application programs issue queries using a declarative query language such as SQL [Melton and

Simon 1993] or OQL [Cattell et al. 1997] and without knowing where and in which format the data is stored in the distributed system. The goal is to execute such queries as efficiently as possible in order to minimize the time that users must wait for answers or the time application programs are delayed. To this end, we will discuss a series of techniques that are particularly effective to execute queries in today's distributed systems. For example, we will describe the design of a query optimizer that compiles a query for execution and determines the best possible way among many alternative ways to execute a query. We will also show how techniques such as caching and replication can be used to improve the performance of queries in a distributed environment. Furthermore, we will cover specific query processing techniques for client-server, middleware (multitier), and heterogeneous database and information systems, which represent architectures that are frequently found in practice.

### 1.2 Scope of this Paper and Related Surveys

A very large body of work in the general area of database systems exists. All this work can be roughly classified into work on architectures and techniques for transaction processing (i.e., quickly processing small update operations), work on query processing (i.e., mostly read operations that explore large amounts of data), and work on data models, languages, and user interfaces for advanced applications. In this paper, we will focus primarily on query processing. A discussion of transaction processing and of alternative data models is beyond the scope of this paper. Transaction processing has been thoroughly investigated in, for example, Gray and Reuter [1993]. Work on data models (relational, deductive, object-oriented, and semistructured) is described in Ullman [1988], Cattell et al. [1997], Abiteboul [1997], and Buneman [1997]. Also, we will assume that the reader is familiar with basic database system concepts, SQL, and the relational data model. Good introductory textbooks are Silberschatz et al. [1997] and Ramakrishnan [1997].

This paper will not even be able to give a full coverage of all query processing techniques used today; in particular, a number of query processing techniques for the World Wide Web are not discussed. For instance, we will not present the architecture of search engines such as AltaVista. Furthermore, there have been several proposals to manage Web sites and query a network of Web pages; see Florescu et al. [1998] for a survey. In addition, several proposals to manage and query XML data exist (e.g., McHugh and Widom [1999], Abiteboul et al. [1999], and Florescu et al. [1999]). Instead of going into the details of all these techniques, the focus of this paper is on fundamental mechanisms to process queries that involve data from several sites. We will, therefore, concentrate on *structured* data (such as that found in relational or object-oriented databases) and on query languages for structured data (such as SQL or OQL). Nevertheless, the techniques described in this paper are also relevant to process other kinds of data in a distributed environment.

A parallel database system is a particular type of distributed system. Distributed and parallel database systems share several properties and goals—in particular, if the parallel system has a so-called "shared-nothing" architecture [Stonebraker 1986]. The purpose of a parallel database system is to improve transaction and query response times, and the availability of the system for *centralized* applications. Parallel systems, therefore, emphasize the cost/scalability arguments described above, while the distributed systems discussed in this paper often address issues such as the heterogeneity of components. While some query processing techniques are useful for both kinds of systems, researchers in both areas have developed special-purpose techniques for their particular environment. In this paper, we will concentrate on the techniques that are of interest for distributed database systems, and will not discuss techniques which are specifically used in parallel database systems (e.g., special

parallel join methods, repartitioning of data during query execution, etc.). An excellent overview on parallel database systems is given in DeWitt and Gray [1992].

In terms of related work, there have been several surveys on distributed query processing; for example, a paper by Yu and Chang [1984] and parts of the books by Ceri and Pelagatti [1984], Özsu and Valduriez [1999], and Yu and Meng [1997] are devoted to distributed query processing. These surveys, however, are mostly focused on the presentation of the techniques used in the early prototypes of the 1970 and 1980. While there is some overlap, most of the material presented in this paper is not covered in those articles and books simply because the underlying technology and business requirements have significantly changed in the last few years.

### 1.3 Organization of this Paper

This paper is organized as follows:

- Section 2. presents the textbook architecture for query processing and a series of basic query execution techniques that are useful for all kinds of distributed database systems
- Section 3. takes a closer look at query processing for one particular and very important class of distributed database systems: client-server database systems
- Section 4. deals with the query processing issues that arise in heterogeneous database systems, that is, systems that are composed of several autonomous component databases with different schemas, varying query processing capabilities, and application programming interfaces (APIs)
- Section 5. shows how data placement (i.e., replication and caching) and query processing interact and shows how data can dynamically and automatically be distributed in a system in order to achieve good performance
- Section 6. describes other emerging and promising architectures for distributed data processing; specifically, this section gives an overview of economic models for distributed query processing and dissemination-based information systems
- Section 7. contains conclusions and summarizes open problems for future research.

## 2. DISTRIBUTED QUERY PROCESSING: BASIC APPROACH AND TECHNIQUES

In this section, we will describe the "textbook" architecture for query processing and present a series of specific query processing techniques for distributed database and information systems. These techniques include alternative ways to ship data from one site to one or several other sites, implement joins, and carry out certain kinds of queries in a distributed environment. The purpose of this section is to give an overview of basic mechanisms that can be used in any kind of distributed database system. In Sections 3. and 4., we will discuss the techniques that are particularly useful for certain classes of distributed database systems (i.e., client-server and heterogeneous database systems).

### 2.1 Architecture of a Query Processor

Figure 1 shows the classic "textbook" architecture for query processing. This architecture was used, for example, in IBM's Starburst project [Haas et al. 1989]. This architecture can be used for any kind of database system including centralized, distributed, or parallel systems. The query processor receives an SQL (or OQL) query as input, translates and optimizes this query in several phases into an executable query plan, and executes the plan in order to obtain the results of the query. If the query is an interactive ad hoc query (dynamic SQL), the plan is directly executed by the query execution engine and the results are presented to the user. If the query is a *canned* query that is part of an application program (embedded SQL), the plan is stored in the database and executed by the query execution engine every time the application program is executed [Chamberlin et al. 1981]. Below is a brief

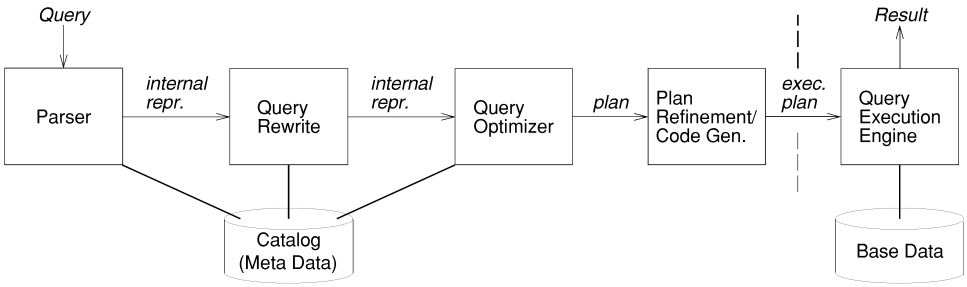



**Fig. 1**. Phases of query processing [Haas et al. 1989].

description of each component of the query processor.

*Parser.* In the first phase, the query is parsed and translated into an *internal representation* (e.g., a query graph [Jenq et al. 1990; Pirahesh et al. 1992]) that can be easily processed by the later phases. The development of parsers is well understood [Aho et al. 1987], and tools like `flex` and `bison` can be used for the construction of SQL or OQL parsers just as for most other programming languages. The same parser can be used for a centralized and distributed database system.

*Query Rewrite.* Query rewrite transforms a query in order to carry out optimizations that are good regardless of the *physical state* of the system (e.g., the size of tables, presence of indices, locations of copies of tables, speed of machines, etc.) [Pirahesh et al. 1992]. Typical transformations are the elimination of redundant predicates, simplification of expressions, and unnesting of subqueries and views. In a distributed system, query rewrite also selects the partitions of a table that must be considered to answer a query [Ceri and Pelagatti 1984; Özsu and Valduriez 1999]. Query rewrite is carried out by a sophisticated rule engine [Pirahesh et al. 1992].

*Query Optimizer.* This component carries out optimizations that depend on the physical state of the system. The optimizer decides which indices to use to execute a query, which methods (e.g., hashing or sorting) to use to execute the operations of a query (e.g., joins and group-bys), and in which order to execute the operations of a query. The query optimizer also decides how much main memory to allocate for the execution of each operation. In a distributed system, the optimizer must also decide at which site each operation is to be executed. To make these decisions, the optimizer enumerates alternative *plans* (described below) and chooses the best plan using a cost estimation model. Almost all commercial query optimizers are based on dynamic programming in order to enumerate plans efficiently. Dynamic programming and considerations for cost estimation in a distributed system are described in more detail in Section 2.2.

*Plan.* A plan specifies precisely how the query is to be executed. Probably every database system represents plans in the same way: as trees. The nodes of a plan are operators, and every operator carries out one particular operation (e.g., join, group-by, sort, scan, etc.). The nodes of a plan are annotated, indicating, for instance, where the operator is to be carried out. The edges of a plan represent consumer-producer relationships of operators. Figure 2 shows an example plan for a query that involves Tables $A$ and $B$. The plan specifies that Table $A$ is read at Site 1 using an index (the *idxscan(A)* operator), $B$ is read at Site 2 without an index (the *scan(B)* operator), $A$ and $B$ are shipped to Site 0 (the *send* and *receive* operators), $B$ is materialized and reread at Site 0 (the *temp* and *scan* operators), and finally, $A$ and $B$ are joined at Site 0 using a nested-loop join method (the *NLJ* operator). The *send* and *receive* operators *encapsulate* all the communication activity so that all other operators

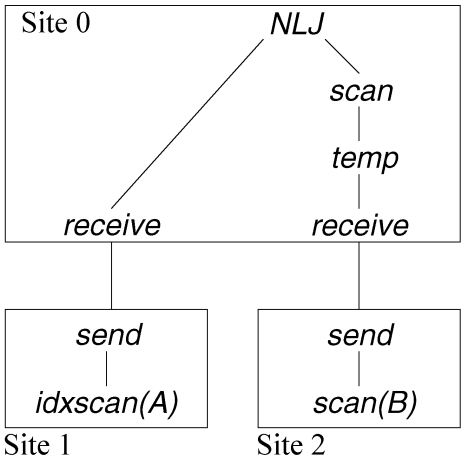



**Fig. 2**. Example query evaluation plan.

(e.g., *NLJ* or *scan*) can be implemented and used in the same way as in a centralized database system.

*Plan Refinement/Code Generation.* This component transforms the plan produced by the optimizer into an *executable plan*. In System R, for example, this transformation involves the generation of an assembler-like code to evaluate expressions and predicates efficiently [Lorie and Wade 1979]. In some systems, plan refinement also involves carrying out simple optimizations which are not carried out by the query optimizer in order to simplify the implementation of the query optimizer.

*Query Execution Engine.* This component provides generic implementations for every operator (e.g., *send*, *scan*, or *NLJ*). All state-of-the-art query execution engines are based on an *iterator model* [Graefe 1993]. In such a model, operators are implemented as iterators and all iterators have the same interface. As a result, any two iterators can be plugged together (as specified by the consumer–producer relationship of a plan), and thus, any plan can be executed. Another advantage of the iterator model is that it supports the pipelining of results from one operator to another in order to achieve good performance.

*Catalog.* The catalog stores all the information needed in order to parse, rewrite, and optimize a query. It maintains the *schema* of the database (i.e., definitions of tables, views, user-defined types and functions, integrity constraints, etc.), the *partitioning schema* (i.e., information about what global tables have been partitioned and how they can be reconstructed), and *physical information* such as the location of copies of partitions of tables, information about indices, and statistics that are used to estimate the cost of a plan. In most relational database systems, the catalog information is stored like all other data in tables. In a distributed database system, the question of *where* to store the catalog arises. The simplest approach is to store the catalog at one central site. In wide-area networks, it makes sense to replicate the catalog at several sites in order to reduce communication costs. It is also possible to cache catalog information at sites in a wide-area network [Williams et al. 1981]. Both replication and caching of catalog information are very effective because catalogs are usually quite small (hundreds of kilobytes rather than gigabytes) and catalog information is rarely updated in most environments. In certain environments, however, the catalog can become very large and be frequently updated. In such environments, it makes sense to partition the catalog data and store catalog data where it is most needed. For example, catalogs of distributed object databases need to know where copies of all the objects (potentially millions) are stored, and they need to update this information every time an object is migrated or replicated. Such catalogs can be implemented in a hierarchical way as described in Eickler et al. [1997].

It should be noted that the architecture shown in Figure 1 and described in this subsection is not the only possible way to process queries. There is no such thing as a perfect query processor. An alternative architecture has, for example, been developed by Graefe and others as part of the Exodus, Volcano, and Cascades projects [Graefe 1995; Graefe and McKenna 1993; Graefe and

**Input:** SPJ query $q$ on relations $R_1, \ldots, R_n$
**Output:** A query plan for $q$

```
1:  for i = 1 to n do {
2:      optPlan({R_i}) = accessPlans(R_i)
3:      prunePlans(optPlan({R_i}))
4:  }
5:  for i = 2 to n do {
6:      for all S ⊆ {R_1, ..., R_n} such that |S| = i do {
7:          optPlan(S) = ∅
8:          for all O ⊂ S do {
9:              optPlan(S) = optPlan(S) ∪ joinPlans(optPlan(O), optPlan(S − O))
10:             prunePlans(optPlan(S))
11:         }
12:     }
13: }
14: return optPlan({R_1, ..., R_n})
```

**Fig. 3**. Dynamic programming algorithm for query optimization.

DeWitt 1987], and is used in several commercial database products (e.g., Microsoft SQL Server 7.0). In that architecture, query rewrite and query optimization are carried out in one phase. Furthermore, there have been proposals to optimize a set of queries rather than individual queries [Sellis 1988]. The advantage of such an approach is that *common subexpressions* (e.g., joins) that are part of several queries need only be carried out once for the whole set of queries.

### 2.2 Query Optimization

We now turn to a description of techniques that can be used to implement the query optimizer of a distributed database system. We will first describe the most popular *enumeration algorithm* for query optimization. After that, we will describe two cost models that can be used to estimate the cost of a plan.

*2.2.1 Plan Enumeration with Dynamic Programming.* A large number of alternative enumeration algorithms have been proposed in the literature; Steinbrunn et al. [1997] contains a good overview, and Kossmann and Stocker [2000] evaluate the most important algorithms for distributed database systems. In the following, *dynamic programming* is described. This algorithm is used in almost all commercial database products, and it was pioneered in IBM's System R project [Selinger et al. 1979]. The advantage of dynamic programming is that it produces the best possible plans if the cost model is sufficiently accurate. The disadvantage of this algorithm is that it has exponential time and space complexity so that it is not viable for complex queries; in particular, in a distributed system, the complexity of dynamic programming is prohibitive for many queries. An extension of the dynamic programming algorithm is known as *iterative dynamic programming*. This extended algorithm is adaptive and produces as good plans as basic dynamic programming for simple queries and "as good as possible plans" for complex queries for which dynamic programming is not viable. We do not describe this extended algorithm in this paper and refer the interested reader to Kossmann and Stocker [2000].

The basic dynamic programming algorithm for query optimization is shown in Figure 3. It works in a bottom-up way by building more complex (sub-) plans from simpler (sub-) plans. In the first step, the algorithm builds an *access plan* for every table involved in the query (Lines 1 to 4 of Figure 3). If Table $A$, for instance, is

replicated at sites $S_1$ and $S_2$, the algorithm would enumerate *scan(A, $S_1$)* and *scan(A, $S_2$)* as alternative access plans for Table *A*. Then, the algorithm enumerates all two-way *join plans* using the access plans as building blocks (Lines 5 to 13). Again, the algorithm would enumerate alternative join plans for all relevant sites, that is, consider carrying out joins with *A* at $S_1$ and $S_2$. Next, the algorithm builds three-way join plans, using access-plans and two-way join plans as building blocks. The algorithm continues in this way until it has enumerated all *n*-way join plans which are complete plans for the query, if the query involves *n* tables.

The beauty of the dynamic programming algorithm is that inferior plans are discarded (i.e., pruned) as early as possible (Lines 3 and 10). A plan can be pruned if an alternative plan exists that does the same or more work at a lower cost. Dynamic programming, for example, would enumerate $A \bowtie B$ and $B \bowtie A$ as two alternative plans to execute this join, but only the cheaper of the two plans would be kept in the *optPlan(A, B)* structure after pruning. Pruning significantly reduces the complexity of query optimization; the earlier inferior plans are pruned, the better because more complex plans are not constructed from such inferior plans.

In a distributed system, neither *scan(A, $S_1$)* nor *scan(A, $S_2$)* may be immediately pruned in order to guarantee that the optimizer finds a good plan. Both plans do the same work, but they produce their results at different sites. Even if *scan(A, $S_1$)* is cheaper than *scan(A, $S_2$)*, *scan(A, $S_2$)* must be kept because it might be a building block of the overall best plan if, for instance, the query results are to be presented at $S_2$. Only if the cost of *scan(A, $S_1$)* *plus* the cost of shipping *A* from $S_1$ to $S_2$ is lower than the cost of *scan(A, $S_2$)*, *scan(A, $S_2$)* is pruned. In general, a plan $P_1$ may be pruned if there exists a plan $P_2$ that does the same or more work and the following criterion holds:

$$\forall i \in \mathit{interesting\_sites}(P_1) : \mathbf{cost}(\mathit{ship}(P_1, i)) \geq \mathbf{cost}(\mathit{ship}(P_2, i)) \quad (1)$$

Here, *interesting_site* denotes the set of sites that are potentially involved in processing the query; the concept is formally defined in Kossmann and Stocker [2000], who also show how this expression can be evaluated efficiently during query optimization under certain conditions. Ganguly et al. [1992] describes further adaptions to the pruning logic that need to be considered if a response time cost model is used (Section 2.2.2).

In the literature, there has been a great deal of discussion concerning bushy or (left-) deep join plan enumeration [Ioannidis and Kang 1991; Lanzelotte et al. 1993; Schneider and DeWitt 1990]. Deep plans are plans in which every join involves at least one base table. Bushy plans are more general; in a bushy plan, a join could involve one or two base tables or the result of one or two other join operations (for instance, the plans of Figure 4 are bushy). The algorithm shown in Figure 3 enumerates all bushy plans, and taking all bushy plans into account is also the approach taken in most commercial database systems. The best plan to execute a query is often bushy and not deep; in particular in a distributed system [Franklin et al. 1996].

*2.2.2 Cost Estimation for Plans. The Classic Cost Model* The classic way to estimate the cost of a plan is to estimate the cost of every individual operator of the plan and then sum up these costs [Mackert and Lohman 1986]. In this model, the cost of a plan is defined as the total resource consumption of the plan. In a centralized system, the cost of an operator is composed of CPU costs plus disk I/O costs. The disk I/O costs, in turn, are composed of seek, latency, and transfer costs. In a distributed system, communication costs must also be considered; these costs are composed of fixed costs per message, per-byte costs to transfer data, and CPU costs to pack and unpack messages at the sending and receiving sites. The costs can be weighted in order to model the impact of slow and fast machines and communication links; for example, it is

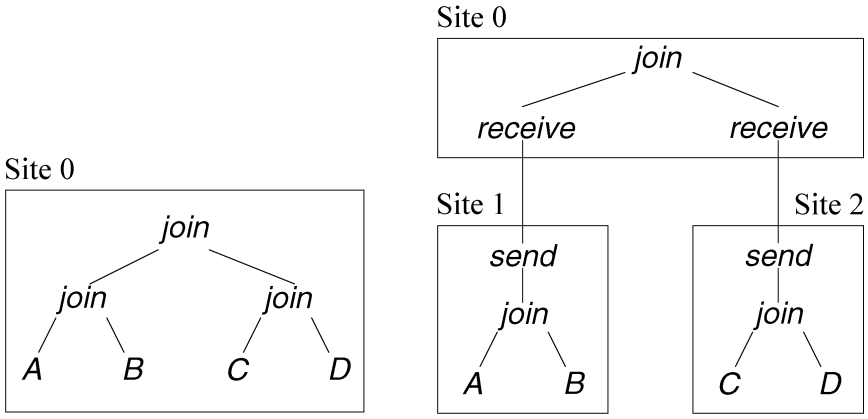



**Fig. 4**. Example plans: total resource consumption vs. response time.

more expensive to ship data from Passau (Germany) to Washington (USA) than from Passau to Munich (Germany). Also, high weights are assigned to the CPU instructions and disk I/O operations that are carried out by heavily loaded machines. As a result, the optimizer will favor plans that carry out operators at fast and unloaded machines and avoid expensive communication links, wherever possible.

*Response Time Models.* The classic cost model that estimates the total resource consumption of a query is useful to optimize the overall throughput of a system: if all queries consume as few resources as possible and avoid heavily loaded machines, then as many queries as possible can be executed in parallel. The classic cost model, however, does not consider intraquery parallelism, so an optimizer based on this cost model will not necessarily find the plan with the lowest response time for a query in cases in which machines are lightly loaded and communication is fast.

To give an example that demonstrates the difference between the total resource consumption and the response time of a plan, consider the two plans of Figure 4. Assuming that the costs of join processing are the same at all three sites and that copies of all tables are stored at all the sites, the first plan clearly has a lower total resource consumption than the second plan because the first plan involves no communication. The second plan, however, probably has a lower response time if communication is fairly cheap because all three joins can be carried out in parallel at the three sites.

To find the plan with the lowest response time for a query (i.e., the second plan of Figure 4), the query optimizer must use a cost model that estimates response time, rather than total resource consumption. Such a cost model was devised in Ganguly et al. [1992]. This cost model differentiates between pipelined and independent parallelism; for example, $A \bowtie B$ and $C \bowtie D$ can be carried out independently in parallel in both plans of Figure 4, and these two joins and the top-level join can be carried out in a pipelined parallel fashion. Described at a high level, this cost model works as follows to deal with both kinds of parallelism (pipelining is slightly more complex). First, the total resource consumption is computed for each individual operator. Second, the total usage of every shared resource used by a group of operators that run in parallel is computed; for example, the usage of the network is computed by taking into account the bandwidth of the network and the volume of data transmitted to carry out all the operators that run in parallel. The response time of an entire group of operators that run in parallel is then computed as the *maximum* of the total

resource consumption of the individual operators and of the total usage of all the shared resources.

To illustrate, let us go back to the two plans of Figure 4 and make the following assumptions: (1) all three joins run in parallel (pipelined and/or independently) in both plans; (2) each join costs 200 sec of CPU time and no disk I/O in both plans; (3) the network has no latency, and shipping the results of $A \bowtie B$ and $C \bowtie D$ each cost 130 sec of network bandwidth in the second plan; (4) sending and receiving tuples incurs no CPU costs; (5) reading all four tables is free in both plans. Under these assumptions, the response time model estimates that the first plan of Figure 4 has a response time of 600 sec; this is the total usage of the CPU at Site 0. For the second plan, the response time model makes the following calculations: total usage of each CPU is 200 sec; total usage of the network is 260 sec; the maximum cost of an operator is 200 sec. As a result, the response time is estimated to be 260 sec, as the *maximum* of all these components.

This cost model captures the effects of operator parallelism in a coarse-grained way; for example, scheduling considerations that arise when several operators concurrently use the same resource are not modeled. Looking closer at the model, it is possible to find situations in which inaccuracies of the cost model make the optimizer choose suboptimal plans even if the resource consumption of the individual operators is accurately estimated. However, the cost model works quite well if only a few operators run in parallel. It has already been successfully used for query optimization in several studies (e.g., Franklin et al. [1996] and Urhan et al. [1998]). Like the classic cost model, it is able to evaluate a plan very quickly. This is important because query optimization often involves applying the cost model to thousands of plans.

### 2.3 Query Execution Techniques

This subsection describes alternative ways to execute queries in a distributed database system. In particular, we will describe how data can be shipped and how joins between tables stored at different sites can be computed. We will not describe "standard" execution techniques that are commonly used in centralized database systems (e.g., hash, sort, or index-based algorithms to compute joins and group-bys). Such techniques have been described in full detail in other surveys [Graefe 1993; Mishra and Eich 1992], and they can naturally be applied in a distributed system in concert with *send* and *receive* operators.

Most of the execution techniques described in this subsection represent one of many options to implement an operator in a distributed system. In order to make the best use of these execution techniques, the query optimizer of the system must be extended in order to decide *if* and *how* to make use of these techniques for a specific query. In other words, integrating these techniques into a distributed database system involves extending the *accessPlans* and *joinPlans* functions in a dynamic-programming-based optimizer (Figure 3) in order to enumerate alternative plans that make use of these execution techniques. Also, cost formulas must be provided so that the cost and/or response time of such plans can be estimated.

*2.3.1 Row Blocking.* As seen in Figure 2, communication is typically implemented by *send* and *receive* operators. Naturally, the implementation of these operators is based on TCP/IP, UDP, or some other network protocol [Tanenbaum 1989]. To reduce the overhead, almost all database systems employ a technique called *row blocking*. The idea is to ship tuples in a blockwise fashion, rather than every tuple individually. In other words, a *send* operator consumes several tuples of its child operator and sends these tuples as a batch. This approach is obviously much cheaper than the naive approach of sending one tuple at a time because the data is packed into fewer messages. The size of the blocks is a parameter of the *send* and *receive* operators; this parameter is set taking into account the characteristics of the network (i.e., the message size of the network).

One particular advantage of row blocking is that it compensates for burstiness in the arrival of data up to a certain point. If tuples are shipped one by one through the network, any short delay in the network would immediately stop the execution of the query at the receiving site because of a shortage of tuples to consume. Due to row blocking, the *receive* operator has a reservoir of tuples and can feed its parent operator even if the next block of tuples is delayed. As a result, it is often better to choose a block size that is larger than the message size used by the network.

*2.3.2 Optimization of Multicasts.* In most environments, networks are organized in a hierarchical way so that communication costs vary significantly depending on the locations of the sending and receiving sites. It is, for instance, cheaper to send data from Munich to Passau, which are both in Germany, than from Washington, across the Atlantic, to Passau. Sometimes, a site needs to send the same data to several sites to execute a query; it is, for instance, possible that the same data must be sent from Washington to Munich and Passau. If the network does not provide cheap ways to implement such multicasts, it is preferable to send the data from Washington to Munich and then forward it from Munich to Passau, rather than sending the data from Washington across the Atlantic twice.

Sometimes, this technique is useful even in a homogeneous and fast network. Let us assume that the time on the wire to send messages between Washington, Munich, and Passau is negligible; in this case, CPU costs to send (i.e., pack) and receive (unpack) messages dominate communication costs. If Washington is heavily loaded or has a slow CPU, then it might again be better if Passau receives the data from Munich rather than from Washington. Obviously, another option would be for Passau to receive the data from Washington and for Munich to receive the data from Passau. The best choice must be made by the query optimizer.

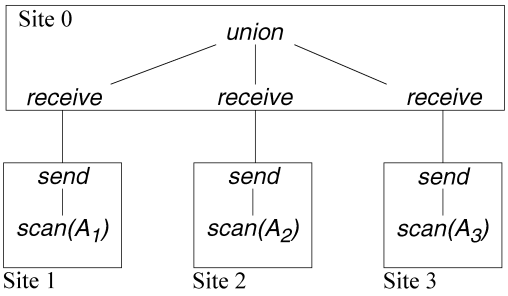



**Fig. 5**. Example union plan.

*2.3.3 Multithreaded Query Execution.* To take the best advantage of intraquery parallelism, it is sometimes advantageous to establish several threads at a site [Graefe 1990]. As an example, consider the plan of Figure 5, which implements the query $A_1 \cup A_2 \cup A_3$; $A_1$ is stored at Site 1, $A_2$ at Site 2, and $A_3$ at Site 3. If the *union* and *receive* operators of Site 0 are executed within a single thread, then Site 0 only requests one block at a time (e.g., in a round-robin way) and the opportunity to read and send the three partitions from Sites 1, 2, and 3 to Site 0 in parallel is wasted. Only if the *union* and *receive* operators at Site 0 run in different threads can the three *receive* operators continuously ask for tuples from the *send* operators at Sites 1, 2, and 3 so that all three *send* operators run and produce tuples in parallel.

Establishing a separate thread for every query operator, however, is not always the best thing to do. First, shared-memory communication between threads needs to be synchronized, resulting in additional cost. Second, it is not always advantageous to parallelize all operations. Consider, for example, the plan of Figure 6 which carries out a sort-merge join of Tables $A$ and $B$. Depending on the available main memory at Site 0, it might or might not be advantageous to *receive* and *sort* Tables $A$ and $B$ in parallel at Site 0. If there is plenty of main memory to store large fractions of both $A$ and $B$ at Site 0, then the two pairs of *receive* and *sort* operators should be carried out in parallel in order to parallelize the *send* and *scan* of $A$ and $B$. Otherwise, the two *receive-sort*

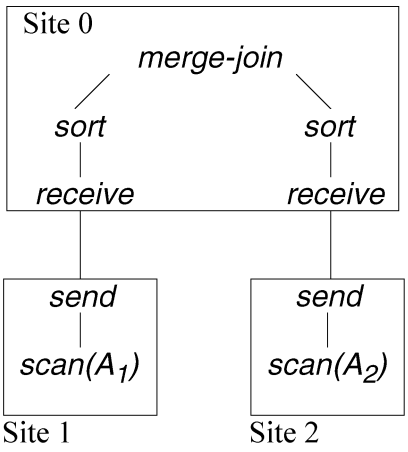



**Fig. 6**. Example join plan.

branches should be carried out one at a time in order to avoid resource contention at Site 0 (i.e., disk thrashing if both sorts write concurrently to the same disk). The query optimizer and/or a scheduler at run time must decide which parts of a query should run in parallel and, thus, which operators should run in the same thread. Work on scheduling and dynamic resource allocation for distributed and parallel databases has been described in, for example, Graefe [1996].

*2.3.4 Joins with Horizontally Partitioned Data.* The logical properties of the *join* and *union* operators make it possible to process joins in a number of different ways if the tables are horizontally partitioned. If, for example, Table $A$ is horizontally partitioned in such a way that $A = A_1 \cup A_2$, then $A \bowtie B$ can be computed in the following two ways [Epstein et al. 1978]:

$$(A_1 \cup A_2) \bowtie B \quad \text{or} \quad (A_1 \bowtie B) \cup (A_2 \bowtie B)$$

If $A$ is partitioned into more than two partitions or if $B$ is also partitioned, then even more variants are possible: for example, $((A_1 \cup A_2) \bowtie B) \cup (A_3 \bowtie B)$ might be an attractive plan if $B$ is replicated and one copy of $B$ is located at a site near the sites that store $A_1$ and $A_2$ and another copy of $B$ is located near the site that stores $A_3$. The optimizer ought to consider all these options.

In some situations, $A$ and $B$ are partitioned in such a way that it is possible to deduce that some of the $A_i \bowtie B_j$ are empty. The optimizer should, of course, take advantage of such knowledge and eliminate such "empty" expressions in order to reduce the cost of join processing. One very common situation is that $A$ and $B$ are partitioned in such a way that $A_i \bowtie B_j$ is empty if $i \neq j$. Consider, for example, a company that has a `Dept` table that is partitioned by `Dept.location` in order to store all the `Dept` information at the site of the department. This company may also have an `Emp` table that is partitioned according to the location of the `Dept` in which the `Emp` works $\texttt{Emp} \bowtie \texttt{Dept}$ can be carried out for this company by joining the `Emp` and `Dept` partitions separately at every site. In other words, the following equation holds if the company has $n$ sites:

$$(\texttt{Emp}_1 \cup \cdots \cup \texttt{Emp}_n) \bowtie (\texttt{Dept}_1 \cup \cdots \cup \texttt{Dept}_n) = (\texttt{Emp}_1 \bowtie \texttt{Dept}_1) \cup \cdots \cup (\texttt{Emp}_n \bowtie \texttt{Dept}_n)$$

*2.3.5 Semijoins.* Semijoin programs were proposed as another technique to process joins between tables stored at different sites [Bernstein et al. 1981]. If Table $A$ is stored at Site 1 and Table $B$ is stored at Site 2, then the "conventional" way to execute $A \bowtie B$ is to ship $A$ from Site 1 to Site 2 and execute the join at Site 2 (or the other way around). The idea of a semijoin program is to send only the column(s) of $A$ that are needed to evaluate the join predicates from Site 1 to Site 2, find the tuples of $B$ that qualify the join at Site 2, send those tuples to Site 1, and then match $A$ with those $B$ tuples at Site 1. Formally, this procedure can be described as follows ($\ltimes$ is the semijoin operator and $\pi(A)$ projects out the join columns from $A$).

$$A \bowtie B = A \bowtie (B \ltimes \pi(A))$$

Variants of this approach are meant to eliminate duplicate tuples from $\pi(A)$ (trading additional work at Site 1 for less communication) and sending a signature file for $A$, called a bloom-hash filter,

rather than $\pi(A)$ [Babb 1979; Valduriez and Gardarin 1984]. Again, the optimizer must decide which variant to use, if any, and in which direction to carry out the semijoin program, from Site 1 to Site 2 or vice versa, based on the cardinalities of the tables, the selectivity of the join predicate(s), and the location of the data used in the other operations of the query.

Experimental work indicates that semijoin programs are typically not very attractive for join processing in standard (relational) distributed database systems because the additional computational overhead is usually higher than the savings in communication costs [Lu and Carey 1985; Mackert and Lohman 1986]. Today, however, several applications that involve tables with very large tuples can be found and semijoin style techniques can indeed be very attractive for such applications. Consider, for example, a table that stores employee information including a picture of every employee. In this case, it does make sense to find the target employees of a query using, say, the *age, dept_no*, and so on columns and then fetch the *picture* and other columns of the query result at the end. Other examples arise in client-server database systems (Section 3.). In a client-server system, for example, the following plan might be very useful

$$(A \bowtie_{S_1} C) \bowtie_{S_3} (B \ltimes_{S_2} C)$$

if $A$ is stored at Server $S_1$, $B$ is stored at Server $S_2$, $C$ is replicated at both servers, and the result of the whole query must be displayed at Client $S_3$ [Braumandl et al. 1999c; Stocker et al. 2001]. Furthermore, Section 4.3.1 demonstrates how semijoin style techniques can be very useful to exploit the specific capabilities of sites in a heterogeneous database system.

*2.3.6 Double-Pipelined Hash Joins.* Recently, double-pipelined (or nonblocking) hash-join algorithms were proposed [Ives et al. 1999; Urhan and Franklin 1999; Wilshut and Apers 1991]. The use of such join algorithms makes it possible to deliver the first results of a query as early as possible. In addition, such join algorithms make it possible to fully exploit pipelined parallelism and thus reduce the overall response time of a query in a distributed system. As described in Urhan and Franklin [1999], variants of such join methods can be particularly useful in a distributed system in which the delivery of tuples through the network is bursty because certain phases of the join processing can be carried out at a site while the site waits for the next, possibly delayed, batch of tuples.

The basic idea on which all these algorithms are based is quite simple. To execute $A \bowtie B$, two main-memory hash tables are constructed: one for tuples of $A$ and one for tuples of $B$. The two hash tables are initially empty. The tuples of $A$ and $B$ are processed one at a time. To process a tuple of $A$, the $B$ hash table is probed in order to find $B$ tuples that match this $A$ tuple; $A$ and the matching $B$ tuples are immediately output. After that, the $A$ tuple is inserted into the $A$ hash table for matching $B$ tuples that have not yet been processed. $B$ tuples are processed analogously. The algorithm terminates when all $A$ and $B$ tuples have been processed and is guaranteed to find all the results of the join. Special actions need to be taken if the hash tables grow in such a way that the main memory is exhausted. To remedy such a situation, the algorithms in Ives et al. [1999] and Urhan and Franklin [1999] adopt a hybrid hashing and partitioning scheme.

*2.3.7 Pointer-Based Joins and Distributed Object Assembly.* One particular kind of query that can be found in object-oriented and object-relational database systems are called pointer-based joins. Pointer-based joins occur because foreign keys are implemented in these systems by explicit references that contain the address of an object or the address of a placeholder of an object [Eickler et al. 1995]. Rather than a user-defined `department_number`, for example, every `Emp` tuple contains a reference or pointer to the site and storage location of the corresponding `Dept` object.

A pointer-based join query is a query that involves traversing a set of references as in "find the Dept information of all Emps that are older than 50 years old."

Alternative ways to execute pointer-based joins have been studied in Shekita and Carey [1990]—that paper focuses on centralized database systems, but the basic ideas can naturally be applied to distributed and parallel database systems [DeWitt et al. 1993]. The naive way to execute pointer-based joins is to scan through the Emp table and follow the Dept references of all Emps of age > 50. In a centralized database system, this naive approach is very expensive because it involves a great deal of random disk I/O to fetch the individual Dept objects from the disk. In a distributed database system, the naive approach incurs even higher costs because it involves a round-trip message to chase the Dept reference of every *old* Emp in addition to random disk I/O. An alternative to the naive approach is to implement the pointer-based join as an ordinary (relational) value-based join; that is, as a join between the Emp and Dept tables with Emp.DeptRef = Dept.address as the join predicate. This approach works if it is known that the Emp tuples only reference objects of the Dept table (i.e., *scoped* references), and this approach typically outperforms the naive approach because it avoids random disk I/O and excessive round trip messages. On the negative side, however, this approach does not take advantage of the fact that the Dept references in the Emp tuples actually materialize which Emps and Depts belong together, and thus, the "value-based join" approach needs to recompute this matching.

The advantages of the "naive" and "value-based join" approaches can in many cases be combined by grouping the Emp tuples using sorting or hashing [Shekita and Carey 1990]. That is, all *old* Emps that belong to Depts stored at the same site are grouped together. Then the Dept objects for these Emps are fetched from that site in one batch. Like the naive approach, this approach does not recompute the matching between Emp and Dept objects, and like the value-based approach, this approach avoids random I/O and unnecessary round trip messages. Random disk I/O can be avoided by sorting the Dept references. Another algorithm, the $P(PM)^*M$ algorithm, to implement pointer-based joins was devised in Braumandl et al. [1999a]. The $P(PM)^*M$ algorithm uses the same partition-based (i.e., grouping) approach as proposed by Shekita and Carey [1990], but the $P(PM)^*M$ algorithm also makes sure that after the pointer-based join is complete, the Emp tuples are in the same order as before. This is useful, for example, if the Emp tuples are already in the right order as needed for the query result, another join operation, or a group-by operation. The $P(PM)^*M$ algorithm is particularly useful if the pointer-based join is along reference sets because it avoids the costs of unnesting the reference sets before the join and then regrouping the sets again after the join.

A special class of algorithms, *object assembly*, becomes attractive if a query involves several pointer-based joins or tries to compute the transitive closure of one or several root objects. Such queries are beginning to become more and more important in the context of the WWW: consider, for example, Web crawlers that recursively traverse references (http links) of Web pages, or systems for semistructured data (e.g., XML data). Traditional join processing takes a breadth-first search approach to evaluate queries with several pointer-based or ordinary joins. The traditional way would be to order the joins during query optimization and execute the joins in the specified order. Object assembly takes a different approach combining breadth-first and depth-first search in a flexible way. Using object assembly in a distributed system, a query involving Emps, Depts, and Divisions, for example, could be executed as follows [Keller et al. 1991; Maier et al. 1994]:

1. Group Emps such that the corresponding Depts referenced by a group of Emps are stored at the same site
2. Consider the first group of Emps and visit the site that stores the Depts for that group of Emps; at that site, fetch all

the referenced Dept objects and, if any, also fetch all Division objects that are stored at that site and referenced by the Dept objects; return the Dept and Division objects

3. At the original site, the site of the Emp objects, Emp-Dept-Division triplets can be directly output; Emp-Dept pairs need to be further expanded by grouping them in such a way that the Divisions referenced by a group of Emp-Dept pairs are stored at the same site (i.e., the grouping of Emp-Dept pairs is carried out just as the grouping of Emps in Step 1, and a group of Emp-Dept pairs can be expanded just as in Step 2).
4. Repeat Steps 2 and 3 until all Emp and Emp-Dept groups have been expanded.

As described in Maier et al. [1994], many variants of this approach are conceivable in a distributed system; unfortunately, none of these variants has been implemented, so experimental performance results are not available.

*2.3.8 Top N and Bottom N Queries.* *Top N* and *Bottom N* queries are another particular kind of query. Examples are "find the ten highest paid employees that work in a research department" or "find the ten researchers that have published the most papers." The goal is to avoid wasted work when executing these queries by isolating the top $N$ (or bottom $N$) tuples as quickly as possible and then performing other operations (sorts, joins, etc.) only on those tuples. In standard relational databases, *stop* operators can be used to isolate the top $N$ and bottom $N$ tuples. Query optimization issues and *stop* operator implementation issues have been discussed in Carey and Kossmann [1997; 1998]. The techniques proposed in this work have been developed primarily for centralized relational database systems, but they can again be directly applied to distributed databases as well. To give a very simple example of how these techniques could be used in a distributed system, consider the plan shown in Figure 7. The given plan computes the top ten tuples

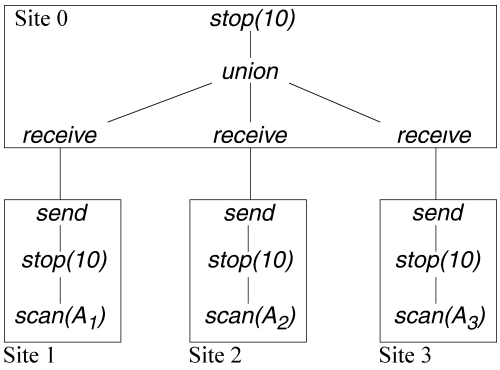



**Fig. 7**. Example plan for a top $N$ query.

of Table $A$ if $A$ is horizontally partitioned over three sites. The *stop* operators at Sites 1, 2, and 3 make sure that every site ships at most ten tuples to Site 0, and the *stop* operator at Site 0 makes sure that no more than ten query results are produced.

Different algorithms need to be used in multimedia database systems [Chaudhuri and Gravano 1996; Fagin 1996] or for meta-searching [Gravano and Garcia-Molina 1997; Gravano et al. 1997]. As an example, consider a query that asks for "ten different kinds of birds that have black feathers and a high voice" using an image database that stores pictures of birds and a sound database that stores recordings of birds' singing. In fact, this query is a *top 10* query because the image and sound databases are *fuzzy*: rather than returning a set of recordings with high voices, the sound database system assigns a *score(voice)* to every recording indicating how high the voice of the corresponding bird is, and it returns the recordings in descending order of *score(voice)*. In the same way, the image database returns pictures of birds in descending order of *score(looks)* that indicates how black the corresponding bird is. The top ten birds are then determined by an overall *scoring function* that computes the total score of a bird; in this case, *min*{*score(voice), score(looks)*} would be an appropriate overall scoring function. Other scoring functions have been described and discussed in Fagin [1996] and Fagin and Wimmers [1997]. The goal is to evaluate such a query in

such a way that the number of images and recordings probed and returned by the image and sound databases is minimized. If the overall scoring function is *min* or any other "monotonic" function,[1] then this task can be done using the following algorithm devised in Fagin [1996]:

1. Continuously ask the image and sound databases for the bird with the next highest (component) score until the *intersection* of the sets of birds returned by the two databases contains at least ten birds.
2. Probe the image and sound databases to evaluate the overall scoring function for all the birds which were returned by one but not both of the two databases in the first step.

This simple algorithm works because the *top 10* birds are within the *union* of the two sets of birds returned by the two databases in the first step: every other bird has definitely lower overall score than the ten birds of the *intersection*, if the scoring function is monotonic. The second step is necessary because the ten birds of the *intersection* are not necessarily the overall winners; it is possible, for example, for a bird that is very black and has a mediocre voice to be among the overall *top 10*, but not in the intersection because of its mediocre voice.

The algorithm above can easily be extended to more than two databases. Similar and slightly more complicated algorithms have been proposed for meta-searching in the WWW. In this environment users are interested in combining the scores for Web pages returned by search engines such as AltaVista, Infoseek, or Lycos in order to find Web pages with a high total score according to all search engines. The algorithm above is not applicable in this environment, and different algorithms are necessary because the second step of the above algorithm (i.e., probing) cannot be carried out using today's WWW search engines [Gravano and Garcia-Molina 1997].

[1] A scoring function $f$ is defined as monotonic if $s_1(a) < s_1(b) \wedge s_2(a) < s_2(b)$ implies that $f(s_1(a), s_2(a)) < f(s_1(b), s_2(b))$.

## 3. CLIENT-SERVER DATABASE SYSTEMS

We now turn to specific classes of distributed systems: systems with a client-server architecture. We will first characterize different kinds of client-server systems and then deal with one of the crucial questions for query-processing in these systems: if and how to exploit the resources of client machines. We will then discuss query optimization and query execution issues, and present several techniques that are popular for query processing in a client-server environment. Some of the techniques presented in this section are also applicable to other system architectures. These techniques are presented in this section because they are mostly used by client-server database systems.

### 3.1 Client-Server, Peer-to-Peer, and Multitier Architectures

In general, *client-server* (or *master-slave*) refers to a class of protocols that allows one site, the client, to send a request to another site, the server, which sends an answer as a response to this request [Tanenbaum 1992]. Using this mechanism, it is possible to implement a variety of different database architectures.

*Peer-to-peer.* This is the most general architecture. In peer-to-peer systems every site can act as a server that stores parts of the database and as a client that executes application programs and initiates queries.

*(Strict) client-server.* In a strict client-server system every site has the fixed role of always acting either as a client (*query source*) or as a server (*data source*). In such a strict client-server architecture, not all the sites can communicate with each other: typically, two clients do not interact and often servers do not interact either.

*Middleware, multitier.* In such an architecture, the sites are organized in a hierarchical way. Every site plays the role of a

server for the sites at the upper level and the role of a client for the lower-level sites. Thus, a site in one of the middle tiers can only communicate with its clients at the level above or its servers at the level below; typically, a site cannot communicate with sites at the same or any other level.

Many examples for distributed database systems with these kinds of architecture can be found. SHORE [Carey et al. 1994] is an example of a system with a peer-to-peer architecture; SHORE is an experimental distributed database system developed at the University of Wisconsin. Most commercial database systems today have a strict client-server architecture. Compared to a peer-to-peer architecture, one advantage of a strict separation between client and server machines is that only server machines need to be administered (i.e., backed up). Also, security issues can be addressed by controlling the server machines and the client-server communication links. Another advantage is that client and server machines can be equipped according to their specific purposes. Client machines are often PCs with good support for graphical user interfaces whereas server machines are usually more powerful with multiple processors, large disks (possibly RAID), and very good I/O performance. An example for a three-tier middleware system is an Intranet with clients running a WWW browser and one or several WWW servers that are connected to database back-end servers. Another example of a middleware system is SAP R/3 [Buck-Emden and Galimow 1996]. SAP is the market leader for business application software (ERP). SAP R/3 installations consist of at least three tiers: (1) presentation servers, which drive the GUIs of the users' desktops, (2) application servers, which implement the business application logic, and (3) database back-end servers, which store all the data. Integrating functionality from different vendors is one reason to use a middleware architecture (i.e., different functionality is provided at different layers of the system). Scalability can be another reason to use a middleware architecture: at every tier, additional sites (i.e., processors) can be added in order to deal with a heavier load.

In the remainder of this section, we will describe query processing techniques that are applicable for all three architectures. For easier presentation and to avoid confusion with the terms client and server, we will concentrate on the strict client-server architecture and assume that every site has the fixed role of acting either as a client or as a server while processing a query. Nevertheless, all techniques are applicable to all three architectures because all three architectures are based on the same paradigm in which query sites and data sites can be different.

### 3.2 Exploiting Client Resources

The essence of client-server computing is that the database is persistently stored by server machines and that queries are initiated at client machines. The question is whether to execute a query at the client machine at which the query was initiated or at the server machines that store the relevant data. In other words, the question is whether to move the query to the data (execution at servers) or to move the data to the query (execution at clients). Another related question is whether and how to make use of caching (e.g., to temporarily store copies of data at client machines). In this section we will present and discuss the trade-offs between alternative approaches which are commonly used in existing systems today.

*3.2.1 Query Shipping.* The first approach is called *query shipping*. Query shipping is used in many relational and object-relational database systems today (e.g., IBM DB2, Oracle 8, and Microsoft SQL Server). The principle of query shipping is to execute queries at servers (i.e., at the lowest level possible in a hierarchy of sites). Figure 8 illustrates query shipping in a system with one server. A client ships the SQL (or OQL) code of a query to the server; the server evaluates the query and ships the results back to the client. In systems with several servers, query shipping works only if there is a middle-tier

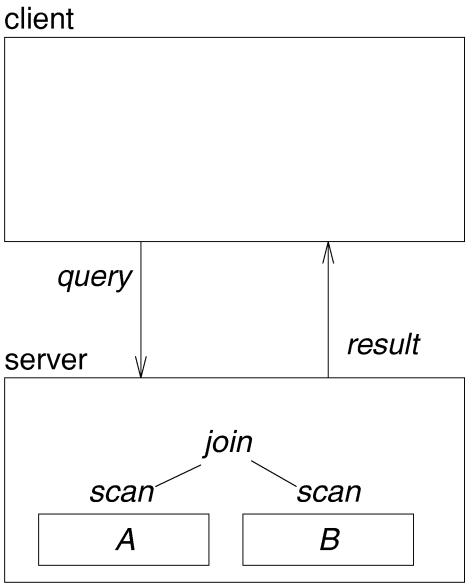



**Fig. 8**. Query shipping.

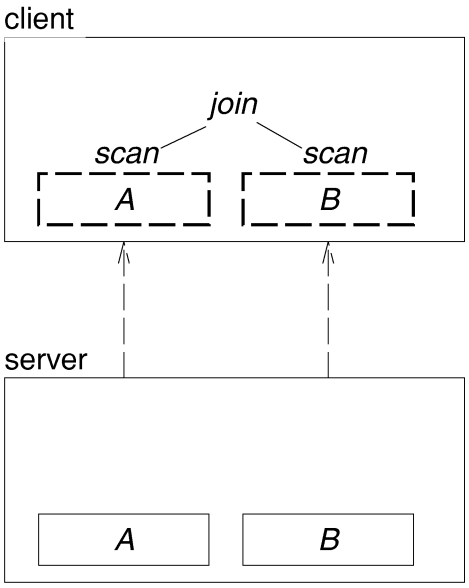



**Fig. 9**. Data shipping.

site that carries out joins between tables stored at different servers or if there are gateways between the servers so that intersite joins can be carried out at one of the servers.

*3.2.2 Data Shipping.* The exact opposite of query shipping is *data shipping*, which is used in many object-oriented database systems (e.g., ObjectStore and $O_2$). In this approach, queries are executed at the client machine at which the query was initiated and data is rigorously cached at client machines in main memory or on disk [Franklin et al. 1993]. That is, copies of the data used in a query are kept at a client so that these copies can be used to execute subsequent queries at that client. Caching is typically carried out in the granularity of pages (i.e., 4K or 8K blocks of tuples) [DeWitt et al. 1990],[2] and it is possible to cache individual pages of base tables and indices [Lomet 1996; Zaharioudakis and Carey 1997]. To illustrate data shipping, consider the example shown in Figure 9, where some pages of Tables $A$ and $B$ are already cached at the client (represented by the dashed boxes in the figure). The *scan* operators at the client use these cached copies of pages and *fault in* all the pages of $A$ and $B$ that are not cached.

*3.2.3 Hybrid Shipping.* Neither data shipping nor query shipping is the best policy for query processing in all situations. The advantages of both approaches can be combined in a *hybrid shipping* architecture [Franklin et al. 1996]. Hybrid shipping provides the flexibility to execute query operators on client and server machines, and it allows the caching of data by clients. The approach is illustrated in Figure 10, where the *scan(A)* and *join* operators are carried out at the client whereas the *scan(B)* operator is carried out at the server. The *scan(A)* operator uses the client's cache as much as possible and ships to the client only those parts of $A$ that are not in the cache. In contrast, the *scan(B)* operator neither uses nor changes the state of the client's cache. (Section 5. contains more information about the impact of query operators on caching.) Today, hybrid shipping is used in some database products such as UniSQL [D'Andrea and

[2] Caching in the granularity of individual tuples, for example, has been studied in Kemper and Kossmann [1994].

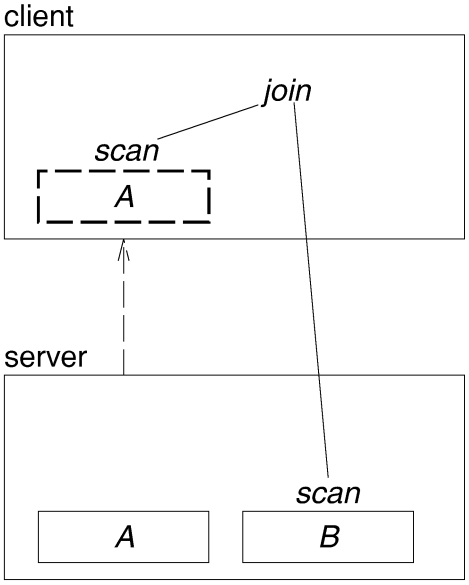



**Fig. 10**. Hybrid shipping.

Janus 1996], application systems such as SAP R/3, database research prototypes such as ORION-2 [Jenq et al. 1990] and KRISYS [Dessloch et al. 1998], and to some extent, in heterogeneous systems such as Garlic [Carey et al. 1995], Mind [Dogac et al. 1996], TSIMMIS [Papakonstantinou et al. 1995a], and DISCO [Tomasic et al. 1998] (Section 4.).

*3.2.4 Other Hybrid Shipping Variants.* For application programs that carry out SQL-style queries and C++-style methods, one special and restricted variant of hybrid shipping is to execute the SQL-style queries at the servers, without caching, and the C++-style methods at the clients, using caching. Such an approach has been proposed, for example, as part of the KRISYS project [Härder et al. 1995]. Persistence is a product that supports this approach [Keller et al. 1993]. This approach is reasonable because caching and client-side execution are particularly effective for methods that repeatedly access the same objects in order to carry out complex computations. Queries that involve a great deal of data, on the other hand, can often be executed more efficiently at server machines without making use of client-side caching.

Another variant of hybrid shipping is used by certain decision support products (e.g., products by MicroStrategy). These products have a three-tier architecture. The bottom tier is a standard relational database system that stores the database and carries out join processing and other standard relational operations. The middle tier then carries out nonstandard operations for decision support like (moving averages, roll-up, drill-down, etc.) [Gray et al. 1996; Kimball and Strehlo 1995]. Again, such an architecture is a special hybrid shipping variant because query processing is carried out at servers and at middle-tier machines, and the difference from full-fledged hybrid shipping is that not all operations can be carried out at all the machines/tiers.

*3.2.5 Discussion.* The performance tradeoffs of query, data, and hybrid shipping have been studied in Franklin et al. [1996]. Many of the effects are obvious. Query shipping performs well if the server machines are powerful and the client machines are rather slow. On the negative side, query shipping does not scale well if there are many clients because the servers are potential bottlenecks in the system. Data shipping scales well because it uses the client machines, but data shipping can be the cause of very high communication costs if caching is not effective and a great deal of unfiltered base data must be shipped to the clients. Obviously, hybrid shipping has the potential at least to match the best performance of data shipping and query shipping by exploiting caching and client resources such as data shipping if that is beneficial, or otherwise by behaving like query shipping. In some situations, hybrid shipping will show better performance than both data and query shipping by exploiting client and server machines and intraquery parallelism to execute a query. The price for this improved flexibility is that query optimization is significantly more complex in a hybrid shipping system

**Table I.** Site Selection Options for Data, Query, and Hybrid Shipping

| | Data Shipping | Query Shipping | Hybrid Shipping |
|---|---|---|---|
| display | client | client | client |
| update | client | server | client or server |
| binary operators (e.g., join) | consumer (i.e., client) | producer of left or right input | consumer or producer of left or right input |
| unary operators (e.g., sort, group-by) | consumer (i.e., client) | producer | consumer or producer |
| scan | client | server | client or server |

than in a query or data shipping system because the optimizer must consider more options. The experiments of Franklin et al. [1996] and other studies demonstrate three other less obvious effects for hybrid shipping systems:

- Sometimes it is better to read data from the servers' disks in a hybrid-shipping system even if the data are cached at the client. Consider, for example, a join query that involves two tables that are stored at two different servers and assume that these tables are cached on the client's disk and that the network is fast. The best way to execute this query might be to read both tables from the servers' disks (rather than from the client's disk cache) and to execute the join at the client. This way, reading the data from the servers' disks and join processing with the client's disk(s) do not interfere with each other.
- Sometimes the best strategy to execute a query in a hybrid shipping system involves shipping cached base data or intermediate query results from the client to a server. Such a strategy, for example, is useful in situations in which the data are cached in the client's main memory, the network is fast, and join operations can be carried out most efficiently at the server.
- Transactions that involve several small update operations should be carried out at clients, thereby putting the new versions of tuples into the client's cache. Such an approach, for example, is used extensively by SAP R/3 [Buck-Emden and Galimow 1996; Kemper et al. 1998]. The advantage is that such transactions can be rolled back at clients without affecting the server and that the updates can be propagated to the server in one batch with fairly little overhead [Bogle and Liskov 1994; O'Toole and Shrira 1994]. Transactions that involve updating large amounts of data (e.g., give all `Emps` a 10% salary increase), on the other hand, should be carried out directly at the server(s) that store the affected data. This way, the original `Emp` table need not be shipped from the server to the client and the updated `Emp` table need not be shipped back to the server either.

In all the experiments presented in Franklin et al. [1996], the other hybrid shipping variants described in Section 3.2.4 perform just like query shipping and perform poorly in many situations. In general, these restricted hybrid shipping variants may perform well for some workloads, just like data or query shipping, but only full-fledged hybrid shipping is able to perform well for any kind of workload.

### 3.3 Query Optimization

Having described query, data, and hybrid shipping as fundamentally different approaches for query processing, we will now show how query optimizers for query, data, and hybrid shipping systems can be built and describe several alternative query optimization strategies.

*3.3.1 Site Selection.* From the perspective of a query optimizer, data shipping, query shipping, and hybrid shipping can be modeled by the options they allow for site selection. Every operator of a plan has a *site annotation,* which indicates where the operator is to be executed. Table I shows the possible site annotations for different

classes of query operators and the three alternative approaches. The table shows the possible annotations for client-server and peer-to-peer systems; analogous annotations can be used for multitier systems. In all three approaches, *display* operators that pass the results of `select` queries to application programs obviously need to be carried out at the client which issued the query. For all other operators, the options of the three approaches are different. Data shipping carries out all operators at the client (i.e., at the site at which the data is *consumed*). In contrast, query shipping carries out all the operators at servers (i.e., at sites at which the data is *produced*). Hybrid shipping allows the optimizer to annotate operators in any way allowed by data or query shipping. The special hybrid variant for decision support and OLAP could be characterized by specifying that scans have *server* site annotations, joins and other standard relational operators have *producer* site annotations like query shipping, and all the other operators (e.g., moving average) have *consumer* site annotations like data shipping.

All site annotations are *logical*. A *client* site annotation indicates that the operator is to be carried out by the client that issues the query; such an annotation does not indicate that the operator is carried out by a specific Machine $x$. Likewise, a *consumer* (*producer*) annotation indicates that the operator is carried out at the same site as the operator that processes the operator's results (input). A *server* annotation for a *scan* indicates that the *scan* is carried out at one of the servers that store a copy of the scanned data. A *server* annotation for an *update* indicates that the *update* is carried out at *all* the servers that store a copy of the affected data.[3] These logical site annotations are translated into physical addresses when a plan is prepared for execution. As a result, the same plan can be used to execute a query at different clients so that a query need not be recompiled for every client individually. If there is replication, translating a *server* annotation for a *scan* involves selecting one specific server machine. This selection can be done heuristically (e.g., the server closest to the client) or in a cost-based manner (Section 3.3.3).

*3.3.2 Where and When to Optimize.* There are two questions of particular interest for query optimization in a client-server environment. The first question is *where* a query should be optimized. Hagmann and Ferrari [1986] studied alternative approaches in an environment with many clients and one server. They propose carrying out certain steps of query processing at the client at which a query originates and other steps at the server. For example, parsing and query rewrite could be carried out at the client whereas query optimization and plan refinement could be carried out at the server. This approach makes sense because operations such as parsing and query rewrite can very well be executed at the clients so that they do not disturb the server, whereas steps such as query optimization require a good knowledge of the current state of the system (i.e., the load on the server) and should, therefore, be carried out by the server. In systems with many servers, no single server has complete knowledge of the whole system. In such systems, one server needs to carry out query optimization (e.g., the server located closest to the client). This server needs to either guess the state of the network and other servers based on statistics of the past, or try to discover the load of other servers by asking them for their current load. While *asking* is obviously better than *guessing* in terms of generating good plans, *asking* involves at least two extra messages for every server that is potentially involved in a query.

The second question, which is related to the above question, is *when* to optimize a query. Again, the answer to this question determines the accuracy, in this case the recency, of the information about the state of the system that the optimizer receives. This question arises for *canned* queries, that are part of application programs and evaluated during the execution of an

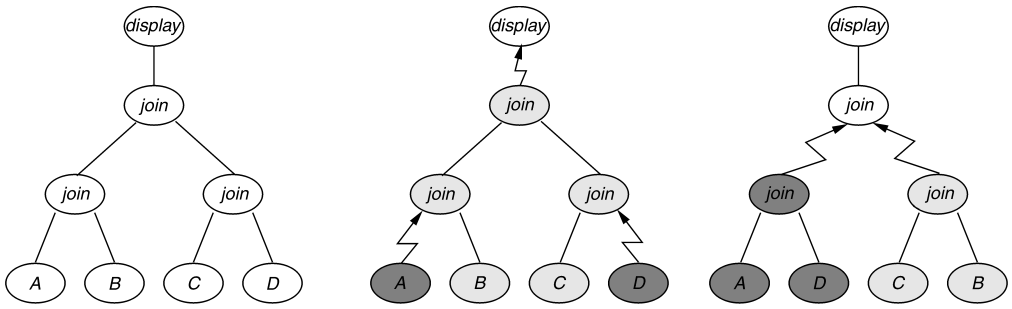
[3] Here, a "read-one-write-all" (ROWA) protocol is assumed.

application program. As already stated in Section 2.1, the traditional approach is to compile and optimize these queries at the time the application program is compiled, store plans for these queries in the database, and retrieve and execute these plans whenever the application program is executed. When something drastic happens that makes the execution of the plan impossible (e.g., when an index used in the plan is dropped), the plan stored in the database is invalidated and a new plan must be generated before the application program is executed [Chamberlin et al. 1981]. Obviously, this approach cannot adapt to changes such as shifts in the load of sites, and the compiled plans show poor performance in many situations.

More dynamic approaches were proposed by Graefe and others in Graefe and Ward [1989], and Cole and Graefe [1994], and by Ioannidis et al. [1992]. The idea is to generate several alternative plans and/or subplans at compile time of the application, store these alternative plans and subplans in the database, and choose the plan or subplans that best matches the current state of the system just before executing the query. Even more dynamic approaches optimize queries on the fly. The idea is to start executing a compiled or dynamically chosen plan and observe whether intermediate query results are produced and delivered at the expected rate. If the expectations are not met, then the execution of the plan is stopped, intermediate results are materialized, and the optimizer is called to find a new plan for those parts of the query that still need to be carried out. Urhan et al. [1998] show how such a reoptimization approach can be very useful to improve the response time of queries in situations in which the arrival of data from certain servers is delayed or bursty because those servers are heavily loaded or the communication links are congested. For this purpose, the approach reorders and reschedules operations at the client so that the client carries out other operations while waiting for the delayed data. In another paper, Kabra and DeWitt show how such a reoptimization approach helps in situations in which the initial plan performs poorly because it was based on wrong estimates of the size of tables and intermediate query results [Kabra and DeWitt 1998].

Ozcan et al. [1996; 1997] proposed another dynamic/on-the-fly query optimization approach. In that approach, queries are optimized and executed in two phases. First, a query is *decomposed*. This means that the query is divided into a set of subqueries that can each be executed by a single server. The final query result is composed by joining the results of the subqueries by the client or a middle-tier machine. Query decomposition for this purpose is described in Evrendilek et al. [1997]. The subqueries are processed by the servers in parallel. The order (i.e., schedule) in which the results of the subqueries are joined at the client depends on the speed in which the servers produce subquery results and the selectivity and cost of the joins which need to be carried out to combine the subquery results. Ozcan et al. propose a heuristic approach to decide whether to join the subquery results produced by two *fast* servers immediately or to delay a join and wait for the delivery of other subquery results from slower servers first. The goal is to parallelize work at the client with work at slow servers as much as possible, as in the reoptimization work of Urhan et al. [1998], and also to avoid the execution of very expensive joins that may result from poor join ordering.

*3.3.3 Two-Step Optimization.* Two-step query optimization is an approach that has become popular for both distributed and parallel database systems [Carey and Lu 1986; Du et al. 1995; Ganguly et al. 1996; Hasan and Motwani 1995; Hong and Stonebraker 1990; Stonebraker et al. 1996; Thomas et al. 1995]. Two-step optimization is an alternative to the dynamic approaches presented in the previous section because it carries out certain decisions just before a query is executed.




(a) 2-Step Plan at Compile Time (b) 2-Step Plan at Run Time (c) Optimal Plan

**Fig. 11**. Increased communication cost due to two-step optimization.

Two-step optimization also reduces the overall complexity of distributed query optimization. Several variants of two-step optimization exist. For distributed systems, the basic variant of two-step optimization works as follows:

1. At compile time, generate a plan that specifies the join order, join methods, and access paths.
2. Every time just before the query is executed, transform the plan and carry out site selection (i.e., determine where every operator is to be executed).

Both steps can be carried out by dynamic programming or any other enumeration algorithm (Section 2.2.1). Two-step optimization has a reasonable complexity because both steps can be carried out with reasonable effort. The first step has essentially the same, mostly acceptable, complexity as query optimization in a centralized database system. The second step also has acceptable complexity because it only carries out site selection. Furthermore, two-step optimization is useful to balance the load on a distributed system because executing operators on heavily loaded sites can be avoided by carrying out site selection at execution time [Carey and Lu 1986]. Two-step optimization is also useful to exploit caching in a hybrid shipping system because query operators can dynamically be placed at a client if the underlying data is cached by the client [Franklin et al. 1996]. On the negative side, two-step query optimization can result in plans with unnecessarily high communication cost. To see why, consider the example shown in Figure 11. The plan in (a) shows the join ordering carried out in the first step of two-step optimization; the plan in (b) shows the result of site selection in the second step; and the plan in (c) shows an optimal plan for this query. In the second and third plans, the site annotations are indicated by the shading of the operators. Tables $A$ and $D$ are colocated at one server (the darkly shaded server), Tables $B$ and $C$ are colocated at another server (the lightly shaded server), and the result of the query must be displayed at a client workstation (the unshaded site). The second plan, obtained using two-step optimization, has a higher communication cost than the optimal plan because the first step of two-step optimization was carried out ignoring the location of data and the impact of join ordering on communication cost in a distributed system.

### 3.4 Query Execution Techniques

Most of the query execution techniques presented in Section 2.3 are useful in a client-server environment as well as in any other distributed database system. Row blocking, for example, is essential to ship data from servers to clients and from clients to servers, and it has been implemented in almost all commercial database systems. Also, it is often attractive to carry out operations at the client in a multithreaded way. In fact, Web browsers like Netscape's Navigator load individual components such as text and

images of a Web page in a parallel and multithreaded way.

One particular issue that arises in hybrid shipping systems is how to deal with transactions that first update data in a client's cache and then execute a query at a server that involves the updated data. For example, consider a transaction that first updates the salary of *John Doe* and then asks for the average salary of all employees. The update is likely to be executed at the client at which the transaction was started in order to batch updates as described in Section 3.2.5. On the other hand, the optimizer will probably decide to execute the second query at the server that stores the `Emp` table in order to avoid the cost of shipping the whole `Emp` table to the client. The point is that the computation of the average salary must consider the new salary of *John Doe*, which is known at the client but not at the server. There are two possible solutions:

- Propagate all relevant updates such as *John Doe's* new salary to the server just before starting to execute the query at the server [Kim et al. 1990].
- Carry out the query at the server and pad the results returned by the server at the client using the new value of *John Doe's* salary—for example, such an approach can be carried out using one of the techniques proposed in Srinivansan and Carey [1992].

In either case, carrying out the query at the server involves additional costs; these additional costs should be taken into account by a dynamic or two-step optimizer in order to decide whether it is cheaper to carry out the query at the server or at the client. Such issues do not arise in query shipping and data shipping systems. Query shipping systems do not support client-side caching and batched updates, and data shipping systems carry out all query operators at the client using the latest cached versions of data.

## 4. HETEROGENEOUS DATABASE SYSTEMS

This section shows how queries can be processed in heterogeneous database systems.[4] The purpose of such systems is to enable the development of applications that need to access different kinds of component databases (e.g., image and other multimedia databases, relational databases, object-oriented databases, or WWW databases). One characteristic of heterogeneous database systems is that the individual component databases can have different capabilities to store data, carry out database operations (e.g., joins and group-bys), and/or communicate with other component databases of the system. For example, a relational database is capable of processing any kind of join whereas a WWW database is typically only capable of processing a specific predefined set of queries. One of the challenges, therefore, is to find query plans that exploit the specific capabilities of every component database in the best possible way and to avoid query plans that attempt to carry out invalid operations at a component database. Another challenge is to deal with *semantic heterogeneity* [Sheth and Larson 1990], which arises, for example, if an application needs the total sales and one component database uses DM as a currency while another component database uses Euro. Furthermore, every component database has its own specific interface (API), decides autonomously when and how to execute a query, and might not be designed to interact with other databases.

There has been a great deal of work on various aspects of the design and implementation of heterogeneous databases. In fact, there have even been excellent tutorials in the past [ACM Computing Surveys 1990], and some commercial systems are described in IEEE Data Engineering Bulletin [1998]. In this section, we will therefore concentrate on basic technology and recent developments in this area. We will present the architecture that is used for most heterogeneous database systems today and discuss how queries can be optimized and executed in heterogeneous systems. Again, keep in mind that we are only

[4] Sometimes, the terms *federated* or *multidatabase system* are used in the same way as we use the term *heterogeneous database system*.

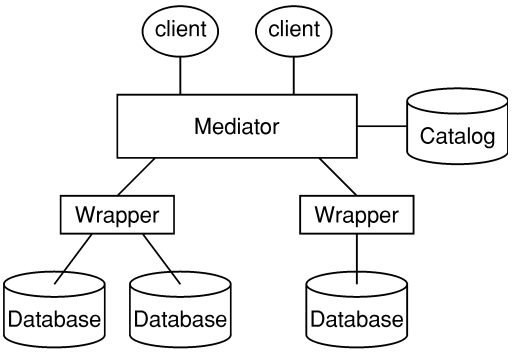



**Fig. 12**. Wrapper architecture of heterogeneous databases.

interested in query processing in this paper. Issues such as transaction processing in heterogeneous database systems are beyond the scope of this paper and have already been described, for example, in Breitbart et al. [1992].

### 4.1 Wrapper Architecture for Heterogeneous Databases

In order to construct heterogeneous database systems, several tools have been developed in recent years; examples are DISCO [Tomasic et al. 1998], Garlic [Carey et al. 1995], Hermes [Adali et al. 1996], TSIMMIS [Papakonstantinou et al. 1995b], Pegasus [Shan et al. 1994], and Junglee's VDB technology [Gupta et al. 1997]. Furthermore, a number of tools have been designed for the specific purpose of integrating data from different relational and object-oriented databases (e.g., IBM's Data Joiner, MIND [Dogac et al. 1996], and IRO-DB [Gardarin et al. 1996]). An older example is HP's MultiDatabase product [Dayal 1983]. Essentially, all of these tools have a three-tier software architecture as shown in Figure 12. Clients connect to a *mediator* [Wiederhold 1993]. The mediator parses a query, carries out query rewrite and query optimization, and executes some of the operations of a query. The mediator also maintains a catalog to store the *global schema* of the whole heterogeneous database system (i.e., the schema used in queries by application programs and users), the *external schema* of the component databases (i.e., which parts of the global schema are stored by each component database), and statistics for query optimization. Thus, the mediator has very much the same structure as the "textbook" query processor described in Section 2.1. The difference is that an extended query optimization approach needs to be used (see Section 4.2) and that certain query execution techniques are particularly attractive in the mediator that might not be attractive in other distributed database systems (see Section 4.3). Also, a mediator is designed to integrate any kind of component database. That is, a mediator does not contain any code that is specific to any one component database and as a result, a mediator cannot directly interact with component databases.

To encapsulate the details of component databases, a *wrapper* (or *adaptor*) is associated to every component database. The wrapper translates every request of the mediator so that the request is understood by the component database's API, and the wrapper also translates the results returned by the component database so that the results are understood by the mediator and are compliant with the external schema of the component database and the global schema of the heterogeneous database. For example, a wrapper of a WWW database (e.g., amazon.com) that returns html pages (e.g., lists of books) must filter out the useful information (e.g., author, title, price, order information) from the html pages. Another example is the wrapper for a sales database that uses DM as currency. This wrapper must convert DM into Euro, if Euro is the currency used in the global schema of the heterogeneous database. In some cases, wrappers also implement special techniques such as row blocking or caching to improve performance. In addition, as described in the next section, wrappers also participate in query optimization.

Obviously, wrappers are fairly complex pieces of software, and it is not unusual for it to take several months to develop a wrapper. The TSIMMIS and Garlic projects have specifically addressed the question of how to make wrapper design

$\texttt{plan\_access}(T, C, P) = \textbf{R\_Scan}(T, C, P, ds(T))$

$ds(T)$ returns the ID of the relational component database that stores $T$.

**Fig. 13**. Access plan enumeration rule for relational component databases.

as cheap as possible [Papakonstantinou et al. 1995b; Roth and Schwarz 1997]. Nevertheless, wrapper development is expensive. The good news is that similar wrappers work for many different kinds of component databases so that it is quite easy to adjust an existing wrapper in order to obtain a wrapper for a new component database. Also, as shown in Figure 12, it is possible for several component databases to be handled by the same wrapper. Furthermore, with the growing importance and demand for heterogeneous systems, it is quite likely that wrappers will be commercially available in the future for many common classes of databases.

One feature of the architecture shown in Figure 12 is that it is extensible. At any time, wrappers and component databases can be upgraded or new component databases can be integrated without changing the mediator or adjusting existing wrappers. Furthermore, the architecture is a *software* architecture. Wrappers and the mediator can be installed at any machines in the system. It is even possible that the mediator is distributed (i.e., that separate cooperating instances of the mediator are installed at different machines).

### 4.2 Query Optimization

This subsection shows how query optimization can be carried out in a heterogeneous database system. As stated at the beginning of this section, one of the challenges of query optimization in a heterogeneous system is that the capabilities of the component databases are different. The optimizer of a heterogeneous system must therefore be generic and be able to *understand* what capabilities component databases have.

Several alternative approaches for query optimization in heterogeneous database systems have been proposed in the literature. One approach is to describe the capabilities of the component databases as views, store the definitions of these views in the catalog, and see during query optimization how a query can be subsumed by the views registered in the catalog [Levy 1999]. While this approach is quite flexible, it is very difficult to implement. Other work has proposed the use of *capability records* [Levy et al. 1996] or context-free grammars to describe the capabilities of queries and the use of various new cost-based and heuristic algorithms to generate plans for a query [Papakonstantinou et al. 1996; Tomasic et al. 1998]. In this section, we will focus on an approach that is based on existing and well-established query optimization techniques. In this approach, the capabilities of the component databases are described by enumeration rules, which are interpreted by the optimizer, and this approach uses either dynamic programming or iterative dynamic programming (Section 2.2.1) in order to find a good plan for a query with reasonable effort. This approach was described in full detail in Haas et al. [1997]. It was implemented for the Garlic system at IBM.

*4.2.1 Plan Enumeration with Dynamic Programming.* The idea is quite simple. Every wrapper provides a set of *planning functions,* which are called by the optimizer's *accessPlan* and *joinPlan* functions in order to construct subplans, (i.e., *wrapper plans*), which can be handled by the wrapper and its component databases. In other words, query optimization is carried out using the same dynamic-programming-based algorithms as described in Section 2.2.1 with the only difference being that the *accessPlan* and *joinPlan* functions call planning functions defined by wrapper developers in order to enumerate subplans rather than constructing such subplans themselves.

Conceptually, planning functions can be seen as *enumeration rules,* and we will

plan_join($S_1, S_2, P$) = **R_Join**($S_1, S_2, P$)
*Condition:* $S_1.Site = S_2.Site$

**Fig. 14**. Join plan enumeration rule for relational component databases.

give several example rules to illustrate the process. Figure 13 shows the plan_access rule of a wrapper for relational component databases. This rule generates an **R_Scan** operator to read table $T$ from the component database that stores $T$ (i.e., *ds(T)*), apply predicates $P$ to the tuples of $T$, and project out columns $C$ of $T$. This rule is called by the optimizer's *accessPlan* function for every table used in a query that is stored by a component database which is associated to the relational wrapper. Consider, for instance, the following query:

```
SELECT e.name, e.salary, d.budget
FROM Emp e, Dept d
WHERE e.salary > 100,000 AND
  e.works_in = d.dno;
```

If Emp and Dept are both stored in the relational component database $D_1$, then the plan_access rule of Figure 13 is instantiated twice as follows:

```
plan_access(Emp, {salary,works_in,name},
      {salary > 100,000}) =
      R_Scan(Emp, {salary,works_in,name},
      {salary > 100,000}, D1)
plan_access(Dept, {dno,budget}, {}) =
      R_Scan(Dept, {dno,budget}, {}, D1)
```

The **R_Scan** operator generated with every application of the plan_access rule is specific to and used internally by the relational wrapper; neither other wrappers nor the mediator need to know about the existence or semantics of such an **R_Scan** operator. Likewise, the relational component databases do not need to know about **R_Scan** operators. To execute plans that involve **R_Scan** operators, the wrapper translates **R_Scan**(*T, C, P, D)* into select C from T where P and submits this query to the relational component database $D$.

Figure 14 shows the enumeration rule that generates join plans for relational component databases. This rule is *called* by the optimizer's *joinPlan* function during join ordering and receives as input two subplans and a set of join predicates. The rule generates a plan with an **R_Join** operator that specifies that the intermediate query results produced by the two subplans should be joined by the relational component database. The rule is only applicable if both subplans are executed by the same relational database; this fact is modeled by the condition $S_1.Site = S_2.Site$. To evaluate this rule the top-level operator of all plans and subplans is annotated as described in Section 3.3.1; for query optimization with heterogeneous data sources, however, the site annotations must always be physical.[5] For the Emp ⋈ Dept example, the rule from Figure 14 would produce the following plan:

```
R_Join(R_Scan(Emp, {salary,works_in,name},
      {salary > 100,000}, D1)
      R_Scan(Dept, {dno,budget}, {}, D1)
      {Emp.works_in = Dept.dno})
```

To execute a plan with an **R_Join** operator, the relational wrapper would translate the plan into an SQL query that involves all the tables and all the join and non-join predicates specified by the operators of the plan.[6]

To give another example, consider the BigBook database on the Web (http://www.BigBook.com). BigBook takes a name or business category and a city or state as input and returns the exact address, telephone number, and so on of all matching businesses. For example, it is possible to ask for all the attorneys in Arkansas. Figure 15 shows the enumeration rules defined by the wrapper for BigBook. All

[5] Other annotations that may be used by rules include the *tables, columns*, and *predicates* involved in a subplan or the *sorting order* in which the top-level operator produces its output [Haas et al. 1997; Lohman 1998].

[6] Precisely, the wrapper would construct the SQL query taking into account the *table, column, predicate*, and *sorting order* annotation of the root operator of the plan.

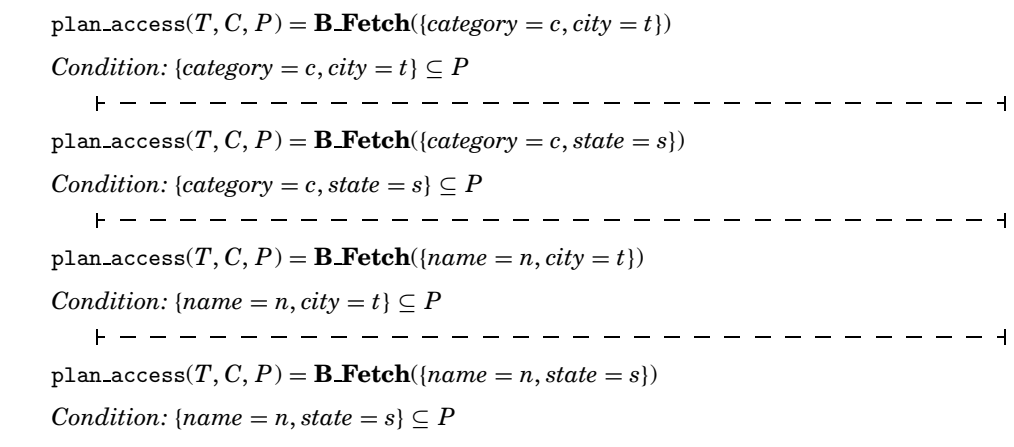



**Fig. 15**. Plan enumeration rule for the BigBook database.

these rules generate a **B_Fetch** operator, which is translated by the BigBook wrapper into an `http` call to `www.BigBook.com`. Independent of the query, the wrapper fetches all the columns (i.e., `name`, `category`, `address`, `telephone`), and depending on the predicates of the query, the wrapper applies a pair of `name/category` and `city/state` predicates. Only `name`, `category`, `city`, and `state` predicates can be applied. To find all the attorneys in Fruitdale Ave., San Jose, the wrapper would generate a plan that returns all attorneys in San Jose (i.e., apply the `category` and `city` predicates) and the *address like "%Fruitdale Ave.%"* predicate would be applied in the mediator. Either a `name` or a `category` predicate can be applied. If a query involves a `name` and a `category` predicate, the enumeration rules of Figure 15 would enumerate two alternative plans, one for each predicate. The optimizer would use the cheaper plan, usually; the other predicate would be applied in the mediator. Furthermore, certain queries cannot be handled although they might be syntactically correct. For example, it is not possible to find all the attorneys in the United State, using the BigBook database because this query is lacking a `city` or `state` predicate. The optimizer would abort processing such a query in a controlled way because the rules of Figure 15 generate no plan to execute such a query. Furthermore, BigBook and its wrapper are not capable of processing joins, so the wrapper provides no *plan_join* rules.

Just like wrappers, the mediator provides a set of rules that enumerate portions of plans that are to be executed by the mediator. For example, the mediator provides a rule to generate plans that apply predicates such as the *address like "%Fruitdale Ave.%"* predicate, which cannot be applied by the component database. The mediator also provides a rule that says that any kind of join can be carried out by the mediator, regardless of where the tables involved in the join are stored. So, an `Emp` ⋈ `Dept` operation could be carried out by the mediator or by the relational component database. The optimizer enumerates both alternatives by calling the mediator and wrapper join enumeration rules, and the overall cheaper plan is selected.

The full details and a description of a more elaborate example can be found in [Haas et al. 1997]. Having presented the basic idea, we will just briefly summarize the major advantages of this approach.

1. This approach relies on well-established distributed database technology; the use of dynamic programming or iterative dynamic programming will generate good plans with reasonable effort just as in any other distributed database system. Using the same technology as most existing database products also gives

vendors an easy migration path to adapt products for heterogeneous database systems.

2. This approach is very flexible so that the capabilities of the component databases can be modeled very accurately. For example, it is possible to write enumeration rules that model gateways between different component databases or replication of tables at different component databases.
3. It is usually fairly easy to implement the enumeration rules of a wrapper. The simple enumeration rules shown in Figures 13 through 15 are actually used in the Garlic project in order to integrate relational databases and Web databases such as BigBook. Enumeration rules and planning functions for wrappers can be very simple because these enumeration rules describe *what kind* of operations can be carried out by a component database rather than exactly *how* these operations are to be carried out.
4. It is possible to define very simple enumeration rules for a new wrapper at the beginning and to add more sophisticated enumeration rules once the wrapper is operational. In fact, some very simple generic rules exist that can be used to integrate any new wrapper and component database [Haas et al. 1997].
5. New wrappers with any kind of enumeration rules can be integrated into the system and the enumeration rules of an existing wrapper can be altered without adjusting the enumeration rules of other wrappers or the mediator and without adjusting any other component of the system.

*4.2.2 Cost Estimation for Plans.* Having described how alternative query evaluation plans can be enumerated in a heterogeneous database system, we now turn to the question of how to estimate the cost or response time of these plans. Both the classic and the response time cost models presented in Section 2.2.2 can be used for this purpose, and the cost or response time of the individual operators that are to be carried out by the mediator can be estimated just as in any other distributed database system because the mediator uses standard, well-understood algorithms to execute joins, group-bys, and so on. The challenge is to estimate the cost or response time of wrapper plans that are to be carried out by the component databases because the details of how a component database executes such a plan (i.e., a subquery) might not be known.

Estimating the cost of wrapper plans in heterogeneous database systems is still an open research issue. There are three alternative approaches, which differ in the accuracy of the estimates and in the amount of required effort by wrapper developers. We will briefly describe these three approaches below. Experiments that demonstrate the importance of accurate cost estimations have been presented in Roth et al. [1999].

*Calibration Approach.* The first approach is called the *calibration approach*. The idea is to define a *generic* cost model for all wrappers and adjust certain parameters of this cost model for every individual wrapper and component database by executing a set of test queries. This way, the specific hardware and software characteristics of a wrapper and a component database can be taken into account. For example, a very simple generic model would be to estimate the cost of a wrapper plan as

$$c * n$$

where $n$ is the estimated number of tuples returned by the wrapper plan (i.e., $n$ depends on the query) and $c$ is the wrapper/component database specific parameter, which would be small for very fast component databases and large for slow component databases or component databases that are only reachable by a slow communication link.

To date, several generic cost models and sample queries have been proposed to implement the calibration approach for heterogeneous databases (e.g., Du et al. [1992], Zhu and Larson [1994], Gardarin et al. [1996], and Roth et al. [1999]).

The generic cost models described in that work are significantly more complex than the simple example we gave above. These cost models typically define special cost formulas for single-table queries, multitable queries, indexed and nonindexed queries, and so on. The big advantage of the calibration approach is that wrapper developers need not worry much about costing issues when they design a new wrapper and/or integrate a new component database into the heterogeneous database. The generic cost model is predefined as part of the mediator, and the calibration of the generic cost model for a new wrapper and component database can be carried out automatically or semiautomatically using the predefined test queries. The big disadvantage of the calibration approach is that not all component databases can be tweaked into a generic cost model. The generic cost models proposed in Du et al. [1992], Zhu and Larson [1994], and Gardarin et al. [1996], for example, are mostly based on observations made with relational or object-oriented database systems, and they are not likely to be a good match for the cost of queries executed, say, by the BigBook database.

*Individual Wrapper Cost Models.* An alternative to the calibration approach is to define a separate cost model for every wrapper. In this approach, the developer of the wrapper not only provides enumeration rules as described in the previous subsection, but also a set of cost formulas. One cost formula is associated with every enumeration rule in order to estimate the cost of the plan(s) generated by that rule. Obviously, the big advantage of this approach is that the cost of all wrapper plans can be modeled as accurately as possible or desired. On the negative side, however, this "do-it-yourself" approach puts a heavy burden on developers of wrappers. To combine the advantages of the calibration approach and this do-it-yourself approach, Naacke et al. [1998] proposed an approach in which costing is done by default using the calibration approach and wrapper developers are free to *overwrite* the default and define their own cost functions for their specific wrappers if they feel that the calibration approach is not sufficiently accurate for their wrappers and component databases. Such a hybrid approach has also been adopted for Garlic [Roth et al. 1999].

*Learning Curve Approach.* The third approach to estimate the cost of wrapper plans is based on monitoring the system and keeping statistics about the cost to execute wrapper plans [Adali et al. 1996]. In this approach, for example, the system would observe that the last three plans that involved Tables $A$ and $B$ had costs of, say, 10 sec, 20 sec, and 9 sec. Based on these statistics, the cost model would estimate that the next plan involving $A$ and $B$ costs 13 sec. Similar and more sophisticated ideas of query feedback have also been studied in the standard relational context [Chen and Roussopoulos 1994]. Like the calibration approach, this approach releases wrapper developers from the burden of worrying about costing issues, but it can be very inaccurate. One particular advantage of this approach is that it automatically and dynamically adapts to changes in the system that impact the cost of operations (e.g., growing tables, hardware upgrades, different load situations).

### 4.3 Query Execution Techniques

We will now discuss two techniques which have become popular for executing queries in heterogeneous database systems. In theory, of course, we would like to take advantage of all the possible ways to execute a query, and many of the basic techniques described in Sections 2.3 and 3.4 are applicable and useful in the mediator of a heterogeneous system (e.g., batching updates or multithreaded query execution). The wrappers and component databases, however, have limited capabilities, which significantly restricts the possible ways to execute a query. For instance, two component databases may not be capable of participating in a semijoin program with duplicate elimination. Also, it is usually not possible to place query operators at

component databases; instead, operators must be translated into queries that are understood by the APIs of the component databases.

*4.3.1 Bindings.* The first technique simulates a nested-loop join in a heterogeneous system. In System $R^*$, a similar technique was called *fetch as needed* [Mackert and Lohman 1986]. This technique exploits the fact that many component databases take input parameters (i.e., *bindings*), as part of their query interfaces. The BigBook database on the Web, for example, takes as input a city and business category and finds the addresses of all matching companies in that city. To illustrate how bindings can be exploited for query processing in heterogeneous systems, consider a heterogeneous system with two relational component databases, $D_1$ and $D_2$, that store Tables $A$ and $B$, respectively. One way to execute $A \bowtie B$ with join predicate $A.x = B.y$ would be as follows:

- the mediator asks $D_1$ to execute the query

  ```
  select * from A
  ```

  in order to *scan* through Table $A$.
- The wrapper of $D_1$ returns tuples of Table $A$ to the mediator, one by one or in blocks using row blocking. For every tuple of Table $A$, the mediator asks the wrapper of $D_2$ to evaluate the following query in order to find the matching $B$'s:

  ```
  select * from B where B.y = ?
  ```

  Here, `?` denotes the binding parameter and is instantiated with the $A.x$ value of the current tuple of $A$.

This approach shows good performance if $A$ is fairly small or a predicate restricts the number of tuples of $A$ that need to be probed. This approach is also useful because it might be the only possible way to execute $A \bowtie B$. BigBook, for example, only allows queries that restrict the business category and city of companies, using predicates with bindings, so that join queries that involve BigBook need to be processed in this way.

Certain component databases accept blocks of tuples as parameters (e.g., relational databases). Such capabilities can be exploited to process joins by passing a block of tuples of the outer table or even the whole outer table to the component database, thereby reducing the number of messages. Adapting the example, the mediator would ask the wrapper of $D_2$ to evaluate the query

```
select * from ? a where B.y = a.x
```

in the second step. Here, `?` is instantiated with a block of tuples from $A$ or the whole $A$ table. Since this blocking reduces the number of messages, it is usually significantly faster than the tuple-at-a-time approach and should therefore always be used if applicable. Blocking corresponds roughly to a block-wise nested-loop join or to a special kind of semijoin program, depending on whether all the columns or only the $x$ column of $A$ are passed to $D_2$.

*4.3.2 Cursor Caching.* There are many workloads for which the mediator submits the same query, with different parameters, many times to a component database. To implement the tuple-at-a-time, binding-based nested-loop join, for example, the same query is submitted for every tuple of $A$. In addition, only four different kinds of queries can be submitted to the BigBook database. The idea of *cursor caching* is to optimize a query only once in order to reduce the overhead of submitting the same query to the same component database repeatedly. For component database systems that understand JDBC [Hamilton et al. 1997], cursor caching can be implemented by using JDBC's `prepareStatement` command to optimize the query, the `set` command to pass the binding parameter(s) every time the query is executed, and the `executeQuery` command to execute the query. Cursor caching is another technique that is extensively used by database application systems such as SAP R/3 [Doppelhammer et al. 1997]. Similar ideas have also been integrated

into several DBMS products (e.g., Oracle8 [Lahiri et al. 1998]).

Cursor caching has the same tradeoffs as static query optimization (Section 3.3.2): on the positive side, cursor caching reduces overhead for query optimization; on the negative side, the (cached) plan might not always be the best plan to execute a query. In particular, the best plan can depend on the value of the query parameter. This effect has been studied for SAP R/3 in Doppelhammer et al. [1997].

### 4.4 Outlook

While query processing for homogeneous and client-server databases is fairly well understood (Sections 2. and 3.), this is not true for heterogeneous systems. Writing wrappers is a tedious task and query optimization is more difficult because the component databases are autonomous, have different capabilities, and incur costs which are hard to predict. Nevertheless, products from database vendors (e.g., IBM's Garlic [Carey et al. 1995] or HP's Pegasus [Shan et al. 1994]) as well as new start-up companies (e.g., Junglee [Gupta et al. 1997]) are already appearing on the market because the management of heterogeneous database systems is extremely important in practice. Furthermore, academic research projects are developing new ways in which database and application components interoperate (e.g., Relly et al. [1998], and Braumandl et al. [1999b]).

This section presented a small fraction of the existing work in this area, and there is definitely a great deal of new work to come. However, this section showed the most important trend. When designing a heterogeneous database, the goal is to encapsulate the heterogeneity of the component databases and to use existing homogeneous distributed database technology as much as possible.

## 5. DYNAMIC DATA PLACEMENT

The previous three sections answered the following question: given a query and the location of copies of data and other parameters, how can this query be executed in the cheapest or fastest possible way. In this section, we will look at this question from a different perspective and show where copies of data should be placed in a distributed system so that the whole query workload can be executed in the cheapest or fastest possible way.

Traditionally, data placement has been carried out statically. With static data placement, a system administrator decides where to place copies of data, speculating what kind of queries might be carried out at what locations in the system. To support static data placement, several models and tools that take the expected query workload and system topology as input and decide where to place copies of data have been devised (e.g., [Apers 1988]). Obviously, static data placement has several weaknesses: (1) the query workload is often not predictable; (2) even if the workload can be predicted, the workload is likely to change, and the workload might change so quickly that the system administrator cannot adjust the data placement quickly enough; (3) the complexity of a sufficiently accurate model for static data placement is too big. (The problem is $\mathcal{NP}$-complete [Apers 1988].) This section is, therefore, focussed on *dynamic data placement* approaches. These approaches keep statistics about the query workload and automatically move data and establish copies of data at different sites in order to adjust the data placement to the current workload. These approaches do not aim to be perfect, but they try to improve the data placement with every move.

As in the rest of this paper, concurrency control and consistency issues are not addressed. Concurrency control issues that are relevant for dynamic data placement have been addressed in, for example, Davidson et al. [1985], Franklin et al. [1997], Lomet [1996], and Zaharioudakis and Carey [1997]. Also, this section only presents techniques that decide where copies of base tables or parts thereof and of indices or parts thereof should be placed. Techniques that place copies of entries of the catalog at different sites are

| | *Replication* | *Caching* |
|---|---|---|
| *target* | server | client or middle-tier |
| *granularity* | coarse | fine |
| *storage device* | typically disk | typically main memory |
| *impact on catalog* | yes | no |
| *update protocol* | propagation | invalidation |
| *remove copy* | explicit | implicit |
| *mechanism* | separate fetch | fault in and keep copy after use |

**Fig. 16**. Differences between replication and caching.

not discussed; such techniques have been specifically studied in Eickler et al. [1997].

### 5.1 Replication vs. Caching

First, we would like to establish some terminology. In principle, there are two different mechanisms to establish copies of data at different sites of a distributed system: replication and caching. Seen from a high level, replication and caching share the same goals: both establish copies of data at different sites in order to reduce communication costs and/or balance the load of a system. As shown in Figure 16, however, there are a number of subtle differences between replication and caching. First of all, replication takes effect at server machines (i.e., data sources) in a client-server environment. That is, replication establishes copies of data at servers based on statistics that are kept at servers with the purpose of better meeting the requirements of a potentially large group of clients. Caching, on the other hand, takes effect at clients or at middle-tier machines (i.e., query sources),[7] and caching is based on statistics kept at these machines. Only one client or a small group of clients, therefore, benefit from a cached copy of a data item, but on the positive side, caching establishes copies of data directly at the places where the data is needed. Also, caching exploits client machine resources which might remain unused without caching (Section 3.2).

The second difference between replication and caching lies in the granularity of the copies of the data. Replication is typically coarse-grained: only a whole table, a whole index, or a whole (horizontal) partition of a table or index can be replicated. Replicating data in a coarse granularity is acceptable because a large group of clients benefit from replication (as stated above), and it is quite likely that most parts of a table or index will be used by this group of clients. Caching, on the other hand, is typically fine-grained: individual pages of a table or index can be cached by a client machine, and some systems even allow the caching of individual rows of a table. Caching in a fine granularity is important because caching supports the queries of a single client or of a fairly small group of clients, and clients tend to be only interested in a small fraction of the data stored in a specific table.

The next four differences listed in Figure 16 are based on the observation that replication decisions are usually more long-term than caching decisions. Again, the background for these differences is that replication is intended to support a large group of clients whose overall access behavior does not change as rapidly as the access behavior of a single client. First, replication typically involves placing data on servers' disks (in part because of the coarse-grained nature of replication), whereas a client's working set of data typically fits in the client machine's main memory.[8] Second, server replicas are

[7] In a multitier environment, caching can be established at all tiers. Caching data at several levels (i.e., hierarchical caching), is also carried as part of the Internet. Furthermore, many institutions provide proxy caches in order to serve a group of clients [Luotonen and Altis 1994].

[8] Note that WWW browsers like Netscape *cache* data on a client's disk, and disk caching has also been shown to be useful in the general database context [Franklin et al. 1993].

registered in the system's distributed catalog so that they can be used by all clients, while caching does not affect the catalog. Third, propagation-based protocols are used to keep replicas of data consistent and accessible at servers at all times. For caching, on the other hand, it was shown that the best way to maintain consistency is to use a protocol that is based on *invalidation* and removes out-of-date copies from a client's cache so that copies of data are only available in a client's cache as long as the data has not been updated [Franklin et al. 1997]. Finally, replicas are kept at servers until they are explicitly deleted whereas copies of data are kept in a client's cache until they are *replaced* by copies of other and more interesting data using a replacement policy such as LRU or until they are removed from the cache because of invalidation.

The last difference between replication and caching concerns the mechanism used to establish copies of data. Replicas are established by a separate process that copies a table, index, or partition and moves it to the target server. Caching, on the other hand, is a by-product of query execution: when a table scan or index scan is executed at a client, the client *faults in* all the pages of the table or index that the client has not cached and, after the scan is complete, the client keeps all the used pages of the table or index in its cache, if the cache is large enough (Section 3.2.2). In other words, replication can occur at servers even if no queries are processed by these servers, whereas the cache of a client is empty if no queries have been processed by that client. As a consequence, caching decisions need to be made by the query processor while replication decisions can be made by a separate component that is established at every server and works independently of the query processor.

Having listed all these differences, one may ask whether one technique is more useful than the other and whether both techniques are needed. We know of no study that answers this question completely, but from the discussion it should have become clear that caching and replication are complementary techniques and that both should be implemented. Replication helps to move data near to a large group of clients so that these clients can access the data cheaply the first time they need the data. Caching makes it possible to access data cheaply when the data are used repeatedly by the same client. In fact, both replication—also called mirroring—and caching are techniques that are frequently used in the WWW and Internet. Another difference between caching and replication is that replication is often used in order to improve the reliability of a system in the presence of server or network failures. Due to its volatile nature, caching cannot serve this purpose. In this section, however, we will concentrate on the performance implications of replication and caching. Finally, *migration* is a particular form of replication in which a new copy is established at the target server and the old copy is removed from the original server.

### 5.2 Dynamic Replication Algorithms

Several dynamic replication algorithms have been proposed in the literature [Bestavros and Cunha 1996; Copeland et al. 1988; Ferguson et al. 1993; Gwertzman and Seltzer 1994; Sidell et al. 1996; Wolfson et al. 1997]. These algorithms can be classified roughly into two groups: (1) algorithms that try to reduce *communication costs* in a WAN by moving copies of data to servers that are located near clients that are likely to use that data, and (2) algorithms that try to replicate *hot* data in order to balance the load on servers in a LAN or in an environment in which communication is cheap (i.e., high bandwidth and low delay). Furthermore, some replication algorithms work particularly well if the network is a tree or has some other simple structure, whereas other algorithms work well in any kind of network. In this subsection, we will briefly describe one specific algorithm, the ADR algorithm [Wolfson et al. 1997], that is targeted to reduce communication costs and works particularly well in tree-shaped networks. The ADR algorithm is a good representative of this class of

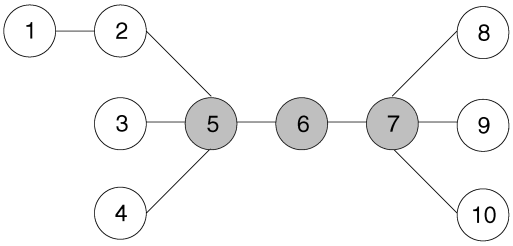



**Fig. 17**. Replication scheme of the ADR algorithm.

algorithms. The ADR algorithm is very simple, has provably good performance in certain environments, and can easily be integrated into most distributed systems. Other replication algorithms that help balance the load of a system and that are based on completely different ideas are presented in Section 6.1.

The ADR algorithm is based on the following observation which holds if a propagation based "read-one-write-all" (ROWA) protocol is used to synchronize updates and keep replicas consistent:

> The replication scheme of an object—table, index, or partition thereof—should be a connected subgraph in order to minimize the communication costs in a tree-shaped hierarchical network.

To illustrate this principle, Figure 17 shows a network with ten servers. In this network, an object is replicated at Servers 5, 6, and 7 (shaded in Figure 17). Even if the object is rarely accessed by the clients of Server 6, the object should nevertheless be replicated at Server 6 if it is replicated at Servers 5 and 7. When the object is updated by a client of Server 5, then this update must be propagated via Server 6 to Server 7 so that the extra copy of the object at Server 6 can be kept consistent without any additional communication cost. Likewise, Server 6's copy of the object can be kept consistent with no additional communication cost if the update originates at a client of Server 7, 8, 9, or 10. If the object is read regardless where, the copy of the object at Server 6 does not hurt either.

Based on this principle, the ADR algorithm expands and contracts the replication scheme of an object at the *borders* of the replication scheme. In the example of Figure 17, Servers 5 and 7 would keep read and write statistics for the object and periodically decide whether the replication scheme should be expanded to Servers 2, 3, 4, 8, 9, or 10, be contracted, removing the replicas at Servers 5 or 7, or remain unchanged. Specifically, Servers 5 and 7 periodically carry out the following tests based on their statistics:

*Expansion Test.* For each of their neighbors that is not part of the replication scheme add the neighbor to the replication scheme if more read requests originate from clients of that neighbor or from clients connected to servers of the subtree rooted in that neighbor than updates originate at other clients. For example, if more read requests originate from clients of Servers 1 and 2 than write requests from clients of all other servers, then Server 2 should be added to the replication scheme.

*Contraction Test.* Drop the copy if more updates are propagated to that copy than the copy is read. If, for example, more updates originate at clients of Servers 6, 7, 8, 9, 10 than read requests originate at Servers 1, 2, 3, 4, 5, then Server 5 should drop its copy of the object.

If the replication scheme consists of only one server, then this server carries out a "switch test" in addition to the expansion test in order to find out whether it might be better to store the only copy of the object at a different server (i.e., carry out migration). Of course, to prevent the only copy of the object from being dropped, the contraction test must not be carried out if the replication scheme consists of only one server.

### 5.3 Cache Investment

We will now turn to caching and a method called cache investment [Kossmann et al. 2000]. Like the ADR algorithm, cache investment keeps statistics and establishes copies of data at clients only if these copies promise to be beneficial. Since replication and caching are different, however, there are a number of important differences between the ADR algorithm and cache investment, and the ADR algorithm is not directly applicable to support caching.

There are two basic ideas behind cache investment. The first idea is to carry out *what-if* analyses in order to decide whether it is worth caching parts of a table or index. More precisely, what-if analyses are applied in order to (1) compute the cost (i.e., *investment*) of loading a client's cache with parts of a table and/or index, and (2) to compute the *benefits* of caching parts of a table or index. The second idea is to extend the optimizer so that the optimizer decides to execute queries at clients if these queries involve data that should be cached at these clients. This way, copies of the data are faulted in at these clients and subsequent queries can be executed using the cache. Queries that involve data that should not be cached should be executed preferably at servers without extra cost for faulting in data.

To illustrate cache investment, consider a client that asks for all `Emps` with `salary > 100,000`:

```
SELECT e.name, e.manager
FROM Emp e
WHERE e.salary > 100,000;
```

Section 3.2.3 mentions that there are essentially two ways to execute this query in a hybrid-shipping system: at the client or at the server. Assuming that there is an index on `Emp.salary` and that the client's cache is initially empty, evaluating this query by using an index scan operator at the client involves faulting in, say, 10 pages of the `Emp.salary` index in order to evaluate the predicate and, say, another 20 pages of the `Emp` table in order to retrieve the `name` and `manager` fields of the `Emps` that qualify. As a result, the overall communication costs are 30 pages if the index scan is carried out at the client. If the index scan is executed at the server, the `name` and `manager` fields of the resulting `Emp` tuples need to be shipped from the server to the client—let's assume a total of 10 pages. As a result, a traditional query optimizer will always decide to execute the index scan at the server.

In this example, cache investment takes effect if the client repeatedly asks queries that involve `Emps` with high salaries. In this case, cache investment advises the optimizer at one point to generate a plan that executes the index scans for these `Emps` at the client. That plan is suboptimal (as described above), but the execution of that plan brings the relevant `Emp` index and table pages into the client's cache so that subsequent queries asking for `Emps` with high salaries can be carried out at the client with no communication cost. Without cache investment, the optimizer would execute all queries at the server, no data would be cached at the client, and every query would involve some communication cost to ship query results from the server to the client. Taking a closer look, cache investment makes the following two calculations for every query issued at a client:

1. The *investment* to load the cache with the relevant index and table pages for highly paid `Emps` is 20 pages for our example query; 20 is the difference in cost between the suboptimal, client-side plan that brings the pages to the client's cache and the optimal, server-side plan. The *investment* might be higher or lower for other queries depending on, among others, the selectivity of the predicates of the `WHERE` clause and the number of columns of the query result.
2. The *benefit* of caching all relevant pages to extract the highly-paid `Emps` is 10 pages for our example query; 10 is the difference in cost between the best plan for the query given that none of the relevant pages are cached, and the cost of the best plan assuming that all relevant pages are cached. Again, the *benefit* of caching might be higher or lower depending on the selectivity of the predicate and the target columns of the query.

As a result of these calculations, cache investment discovers that after three "high-salary" queries, the *benefits* of caching outweigh the *investment*. After three queries, cache investment will thus advise the optimizer to generate a suboptimal plan in order to load the cache with the relevant `Emp` data.

Quite a few more details need to be taken into account to make cache

investment work properly; for instance, the exact interaction of cache investment and query optimization, dealing with updates, limitations in the size of a client's cache, lightweight strategies to estimate the *benefits* and *investment* of caching, cost formulas for clustered and unclustered indices, considering response time rather than communication costs in the calculations, and keeping statistics in the presence of rapidly changing client workloads. All these details have been described in Kossmann et al. [2000], so we will not discuss them here. To conclude, here again are the differences between caching with cache investment and replication with the ADR algorithm:

- Caching is fine-grained, making it possible to cache only a few, frequently used pages of large tables or indices as in the example above. Shipping, caching, and keeping consistent copies of the whole `Emp` table at all the clients that frequently ask for `Emp` information is usually not practical.
- The investment to establish a copy is significantly lower with caching than with replication because caching takes effect when data is read from disk and shipped to the client in order to execute a query. In our example, the investment was 20 pages although 30 pages had to be shipped to the client. Replication always pays the full price of 30 pages (or even more due to its coarse granularity) to establish a copy because the replication process does not overlap with the execution of queries.
- As mentioned in Section 5.1, however, probably both replication with the ADR algorithm (or some other algorithm) and caching with cache investment (or some similar technique) should be used because caching and replication take effect at different "ends" of the system.

### 5.4 View Caching, View Materialization, and Data Warehouses

At the end of this section, we would like to comment on the *kinds* of data that can be cached and replicated. So far, we assumed that only *base data* can be cached and replicated (i.e., base tables or indices or parts of them). We now turn to systems that cache or replicate (i.e., materialize) *derived data* or *views*. Such systems could, for example, cache the average salary of all `Emps` that work in a research department instead of or in addition to the complete `salary` information of all `Emps`.

View caching and materialization has been addressed in a number of research projects (e.g., Roussopoulos et al. [1995], Keller and Basu [1994], Dar et al. [1996]; Deshpande et al. [1998], and Dessloch et al. [1998]). View materialization has also been implemented in Oracle 8 [Bello et al. 1998]. Data warehouses are the most prominent example of commercial systems that materialize and/or cache views [Widom 1995]. Data warehouses are typically established for decision support in companies or as product catalogs and classified ads for electronic commerce on the Web. They are usually installed in a three-tier environment. The data warehouse is located in the middle tier, it is connected to one or more data sources, and it keeps materialized views over the base data stored at those data sources in order to answer queries from clients without interacting with the data sources. In fact, a huge industry has already been formed around this concept, and data warehousing definitely deserves more attention than we give it in this small section. From our narrow perspective, a data warehouse, the data sources, and the clients are part of a distributed system in which views are materialized or cached in the warehouse.

Compared to the replication and caching of base data, the benefits of materializing and caching views are significantly larger. Caching the result of a join or aggregate query, for example, might completely eliminate the cost of join or group-by processing for subsequent queries in addition to savings in communication costs and potential load balancing effects. View caching and view materialization, however, are significantly more complex to implement. First, keeping cached or materialized views consistent in the presence of updates is complex and often expensive

[Roussopoulos 1991; Quass and Widom 1997], and it is unclear how invalidation-based protocols, which have proven to be very useful to implement cache consistency, can be applied to view caching. Second, the ADR algorithm obviously cannot be applied to decide what views to materialize, and algorithms that carry out such decisions are just beginning to emerge [Harinarayan et al. 1996; Scheuermann et al. 1996; Yang et al. 1997]. Cache investment can be used, but there is an explosion in the number of "what-if" analyses that need to be carried out for every query so that a naive application of cache investment is impractical. Third, query optimization is more complicated and more expensive in the presence of cached and/or materialized views [Levy 1999]. The optimizer must determine whether a cached or materialized view is *applicable*—this is known as the *containment* or subsumption problem [Levy 1999]. After that, the optimizer must decide which of the applicable views to use. To this end, the optimizer must be extended in order to enumerate *read(view)* plans for all applicable views just like other *access* and *join plans* and carry out cost-based optimization using dynamic programming or iterative dynamic programming (Section 2.2.1). If, for example, a materialized view involves Tables `Emp` and `Dept` and was shown to be applicable for a query that involves the `Emp`, `Dept`, and `Division` tables, the view can be used as an access plan for the `Emp` table, as an access plan for the `Dept` table, and as a `Emp` $\bowtie$ `Dept` join plan. In other words, the view, if it is applicable, can be seen as a component database that stores copies of the `Emp` and `Dept` tables and is capable of processing joins, and query optimization in the presence of views can be carried out in the same way as query optimization in the presence of heterogeneous component databases as described in Section 4.2.

## 6. NEW ARCHITECTURES FOR DISTRIBUTED QUERY PROCESSING

The previous sections presented a comprehensive set of techniques to implement distributed database and information systems. While this set of techniques is sufficient for most of today's applications, the advent of the Internet has sparked a large number of new applications and led to systems with an ever growing number of clients and servers. In such an environment, the conventional query processing approach presented in the previous sections might be too rigid. In this section, we will describe recent trends and developments. Specifically, we will give a brief overview of economic models for distributed query processing and dissemination-based information systems.

### 6.1 Economic Models for Distributed Query Processing

A large variety of economic models for various aspects of distributed computing have been studied since the mid-1980s (e.g., economic models for resource allocation, load balancing, flow control, and quality of service). A good survey of such techniques can be found in Ferguson et al. [1996]. The motivation to use an economic model is that distributed systems are too complex to be controlled by a single centralized component with a universal cost model. Systems based on an economic model rely on the "magic of capitalism." Every server that offers a service (data, CPU cycles, etc.) tries to maximize its own *profit* by selling its services to clients. The hope is that the specific needs of all the individual clients are best met if all servers act this way.

Mariposa is the first distributed database system based on an economic paradigm [Stonebraker et al. 1996]. Mariposa processes queries by carrying out auctions. In such an auction, every server can bid to execute parts of a query, and clients pay for the execution of their queries. More precisely, query processing in Mariposa works as follows (more details can be found in Stonebraker et al. [1996]):

1. Queries originate at clients, and clients allocate a budget to every query. The budget of a query depends on the importance of the query and how long the

client is willing to wait for the answer. A client in Las Vegas could, for example, be willing to pay $5.00 if the client gets the latest World Cup football results within a second, but only 10 cents if the delivery of the results takes one minute.

2. Every query is processed by a broker. The broker parses the query and generates a plan that specifies the join order and join methods. For this purpose, the broker may employ an ordinary query optimizer for a centralized database system based on, for example, dynamic programming.
3. The broker starts an auction. As part of this auction, every server that stores copies of parts of the queried data or is willing to execute one or several of the operators specified in the broker's plan is asked to give bids in the form of

   $\langle Operator\ o, Price\ p, Running\ Time\ r, Expiration\ Date\ x \rangle$

   In other words, with such a bid a server indicates that it will be willing to execute Operator $o$ for $p$ dollars in $t$ seconds, and that this offer is valid until the expiration date $x$.
4. The broker collects all bids and makes contracts with servers to execute the queries. Doing so, the broker tries to maximize its own profit. If, for example, the broker finds a way to execute the Las Vegas query from above in a second, paying only $1.00 to servers, the broker will pursue this way and keep $4.00 of the budget as profit. If the query cannot be evaluated with acceptable cost in one second, the broker will try to find a very cheap way to execute the query in a minute and keep a couple of cents as profit. If the broker finds no way to execute the query within the time/budget limitations, the broker will reject the query. In this case, the client must raise the budget, revise the response time goals, or just be happy without the answer.

At first glance, Mariposa's query processing approach does not appear to be very different from the techniques presented in Sections 2. and 3.. Mariposa carries out two-step optimization as described in Section 3.3.3, making it possible to avoid heavily loaded or slow servers. The beauty of Mariposa is that different servers can flexibly establish different bidding strategies in order to achieve high revenue. For instance, a server might specialize in high-end or low-end services. Using an example from real life, there are expensive restaurants for people that like to eat well and fast-food restaurants for people with other needs. This diversity makes it possible to meet the eating habits of a large group of people. Mariposa supports such a diversity in the services provided by a distributed database system.

Another advantage of Mariposa is that dynamic data placement fits nicely into Mariposa's economic approach. In addition to the revenue for executing query operators, servers can make a profit by buying and selling copies of data [Sidell et al. 1996]. The soccer WWW server located in Paris, for example, was not able to handle all the requests from all over the world during the World Cup finals in 1998. Using Mariposa, that server could have allowed other servers, say, in Brazil or Nigeria to replicate the results of the soccer matches and have gotten additional revenue for selling the original copy of the `Results` table and for propagating all the updates. Servers in Brazil and Nigeria would have bought copies of the `Results` table to bid for queries that involve that data and/or sell copies of that data to other servers (e.g., in Argentina or Cameroun).

While all these concepts sound very promising and a version of Mariposa is already available commercially (distributed by Cohera), it is still unclear how well Mariposa and other systems with economic models will work in practice. There is a significant amount of research required to find out how to configure the bidding and data buying/selling strategies of servers and how to keep the overheads of the bidding protocols within reasonable limits.

### 6.2 Dissemination-Based Information Systems

Throughout this paper, a *request-driven* data delivery model was assumed. In this model, users or application programs (i.e., clients) are active and initiate queries; servers are passive and process queries upon request. Lately, there has been a great deal of interest in *push technology*. In this model, servers are active and disseminate data to clients before the clients ask for the data. An early incarnation of *push* is TeleText, provided by most European TV stations since the mid-1980s. Furthermore, both Netscape's Navigator and Microsoft's Internet Explorer provide features to allow clients to passively listen to data that is disseminated by WWW servers. Pointcast's screensaver, which displays news and commercials based on a user's profile of interests, is another product in this domain. A good overview of these and other push-based systems is given in Franklin and Zdonik [1998].

One reason for this interest is that many people like to obtain all the information they are interested in with virtually no effort. In addition, there are a number of technical reasons in favor of *push*—in particular, if data is dissiminated in networks that support broadcasts or *1:N* multicasts. Most importantly, push-based systems scale better than traditional request-driven systems. Rather than processing every request individually, push-based systems satisfy the requests of several users by disseminating the results only once [Aksoy and Franklin 1998]. Data push and request-driven access to data can also be combined in order to achieve high scalability and satisfy unusual user requests at the same time [Acharya et al. 1997]. Other interesting aspects are client-side caching in push-based systems [Acharya et al. 1995; 1996], and multitier architectures for data dissemination [Franklin and Zdonik 1998].

Unfortunately, SQL-style query processing has not yet been studied in the context of push-based systems. For example, it is still unclear which of the techniques presented in this paper would be applicable for a push-based system. A great deal of future work remains to be done in this area.

## 7. CONCLUSION

In the last decade, the landscape of distributed database and information systems has changed tremendously. Network technology has become mature, and as a result, businesses rely more and more on distributed and on-line data processing architectures as opposed to monolithic and batch-oriented architectures. In addition, a whole new generation of distributed database applications is appearing, exploiting, for example, the Internet or wireless communication networks for mobile clients. Furthermore, most systems today have a client-server or a multitier architecture, and many complex systems are composed of several subsystems from potentially different vendors with heterogeneous data processing capabilities and APIs.

In this paper an overview of the state of the art in distributed query processing was given. This paper discussed various query processing techniques developed for recent products and research prototypes and showed how they can be applied to different types of distributed systems. Many different architectures and applications can be found today, but all these architectures can roughly be characterized by their communication paths (client-server, peer-to-peer, or multitier) and by the capabilities of the sites of the system (homogeneous or heterogeneous). For each category, the paper presented and discussed that set of query processing techniques which are particularly effective. For instance, the paper showed how to exploit client resources in a strict client-server system and how to exploit the query capabilities of individual sites in a heterogeneous system.

Independent of the specific architecture, all distributed database and information systems today are based on the following two principles:

- *Best effort:* the query processor always tries to execute a query as fast as possible or with as little cost as possible. At the heart of this strategy is a query optimizer, which decides for every query which query execution methods to use (e.g., which join method), where to execute these methods, and in which order to execute these methods. The optimizer can be used *statically* in order to compile a query once and for all times. The optimizer can also be used *dynamically* just before a query instance is executed or *on the fly* while the query is executed in order to adjust to the current state of the system.
- *Flexible data placement:* in order to improve the performance of a whole query workload, caching and/or replication can be used in order to place data at or near sites where the data are frequently used.

Both query optimization and caching/replication are extensively studied in this paper.

Combined, the set of techniques presented in this paper should be sufficient to support most of today's database applications. Nevertheless, there are a number of avenues for future work:

- No vendor has implemented all or even a significant portion of the techniques described in this paper. Conceptually, the pieces fit together well, but it is nevertheless not always easy to integrate a new technique into an existing system. For example, it is possible to extend a query optimizer to consider a new query evaluation algorithm, but doing so might substantially increase the running time of the optimizer. As a result, a tricky compromise must be found that extends the optimizer so that the new algorithm is supported reasonably well and the increase in optimization time is tolerable.
- The techniques described in this paper can be implemented as part of a distributed database management system or as part of a database application system. Preferably, of course, the techniques should be implemented as part of a database management system so that any kind of application can directly benefit from them. In fact, however, several of the techniques presented in this paper have been implemented as part of the SAP R/3 business application system [Kemper et al. 1998] because standard, off-the-shelf database management systems have not yet implemented these techniques. This situation might be the cause for a great deal of confusion, and ultimately certain application systems might not work well with certain database management systems if conflicting techniques are carried out on both ends or important techniques are not carried out at all. Coordinating all the different query processing activities is a difficult task in such systems. The situation is getting worse with the current trend to design and market application and database management modules that can be freely plugged together and may interact in unpredictable ways.
- Scalability remains a major concern. It is still unclear whether query optimization and the "best effort" approach work in a system with ten thousands of servers and millions of clients because so far nobody has been able to simulate query processing in such a large scale. In addition, further research is necessary in order to find out how well economic models and data dissemination models work for large-scale query processing.
- Most of this paper was focused on structured (i.e., relational) data. There is still a great deal of work to be done in order to integrate other types of data (e.g., XML, text, images, etc.). Furthermore, it is important to deal with approximate or partial answers on the Internet; on the Internet, failure is the rule rather than the exception.

## ACKNOWLEDGMENTS

I would like to thank Alfons Kemper for suggesting that I write this paper and for many helpful discussions and comments. I would also like to thank Reinhard Braumandl, Mike Carey, Gerhard Drasch,

André Eickler, Mike Franklin, Laura Haas, Björn Jónsson, and Konrad Stocker—some of the ideas described in this paper come from joint work with them. I am also grateful to Judith Kossmann, the editor, and the anonymous referees whose detailed comments helped greatly to improve the presentation of this paper.

## REFERENCES

ABITEBOUL, S. 1997. Querying semi-structured data. In *Proceedings of the International Conference on Database Theory (ICDT)* (Delphi, Greece, Jan.).

ABITEBOUL, S., BUNEMAN, P., AND SUCIU, D. 1999. *Data on the Web, from Relations to Semistructured Data and XML*. MORKAU, MKADDR.

ACHARYA, S., ALONSO, R., FRANKLIN, M., AND ZDONIK, S. 1995. Broadcast disks: Data management for asymmetric communication environments. In *Proceedings of the ACM SIGMOD Conference on Management of Data* (San Jose, CA, May), 199–210.

ACHARYA, S., FRANKLIN, M., AND ZDONIK, S. 1996. Prefetching from a broadcast disk. In *Proceedings IEEE Conference on Data Engineering* (New Orleans, LA, Feb. 1996), 276–285.

ACHARYA, S., FRANKLIN, M., AND ZDONIK, S. 1997. Balancing push and pull for data broadcast. In *Proceedings of the ACM SIGMOD Conference on Management of Data* (Tucson, AZ, May), 183–194.

ACM Computing Surveys. 1990. Special issue on heterogeneous databases. *ACM Computing Surveys*, *22*, 13.

ADALI, S., CANDAN, K., PAPAKONSTANTINOU, Y., AND SUBRAHMANIAN, V. S. 1996. Query caching and optimization in distributed mediator systems. In *Proceedings of the ACM SIGMOD Conference on Management of Data* (Montreal, Canada, June), 137–148.

AHO, A., SETHI, R., AND ULLMAN, J. 1987. *Compilers: Principles, Techniques and Tools*. Addison-Wesley.

AKSOY, D. AND FRANKLIN, M. 1998. Scheduling for large-scale on-demand data broadcasting. In *Proceedings IEEE INFOCOM Conference* (San Francisco, CA, March).

APERS, P. 1988. Data allocation in distributed DBMS. *ACM Transactions on Database Systems 13*, 3 (Sept.), 263–304.

BABB, E. 1979. Implementing a relational database by means of specialized hardware. *ACM Transactions on Database Systems 4*, 1 (March), 1–29.

BELLO, R. G., DIAS, K., DOWNING, A., JR., J. F., NORCOTT, W. D., SUN, H., WITKOWSKI, A., AND ZIAUDDIN, M. 1998. Materialized views in oracle. In *Proceedings of the Conferencce on Very Large Data Bases (VLDB)* (New York, Aug.), 659–664.

BERNSTEIN, P., GOODMAN, N., WONG, E., REEVE, C., AND ROTHNIE, J. 1981. Query processing in a system for distributed databases (SDD-1). *ACM Transactions on Database Systems 6*, 4 (Dec.), 602–625.

BESTAVROS, A. AND CUNHA, C. 1996. Server-initiated document dissemination for the WWW. *IEEE Data Engeneering Bulletin 19*, 3 (Sept.), 3–11.

BOGLE, P. AND LISKOV, B. 1994. Reducing cross domain call overhead using batched futures. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems and Languages (OOPSLA)* (Portland, OR, Oct.), 341–354.

BRAUMANDL, R., CLAUSSEN, J., KEMPER, A., AND KOSSMANN, D. 2000. Functional join processing. *The VLDB Journal*. *8*, 3–4 (Feb.), 156–177.

BRAUMANDL, R., KEMPER, A., AND KOSSMANN, D. 1999. Database patchwork on the internet (project demo description). In *Proceedings of the ACM SIGMOD Conference on Management of Data* (Philadelphia, PA, June), 550–552.

BREITBART, Y., GARCIA-MOLINA, H., AND SILBERSCHATZ, A. 1992. Overview of multidatabase transaction management. *The VLDB Journal 1*, 2 (July), 181–293.

BUCK-EMDEN, R. AND GALIMOW, J. 1996. *SAP R/3 System, A Client/Server Technology*. Addison-Wesley, Reading, MA.

BUNEMAN, P. 1997. Semistructured data. In *Proceedings of the ACM SIGMOD/SIGACT Conference on Principle of Database Systems (PODS)* (Tucson, AZ, May), 117–121.

CAREY, M., HAAS, L., SCHWARTZ, P., ANYA, M., CODY, W., FAGIN, R., FLICKNER, M., LUNIEWSKI, A., NIBLACK, W., PETKOVIC, V., THOMAS, J., WILLIAMS, J., AND WIMMERS, E. 1995. Towards heterogeneous multimedia information systems. In *Proceedings of the International Workshop on Research Issues in Data Engineering* (March), 124–131.

CAREY, M., DEWITT, D., FRANKLIN, M., HALL, N., MCAULIFFE, M., NAUGHTON, J., SCHUH, D., SOLOMON, M., TAN, C., TSATALOS, O., WHITE, S., AND ZWILLING, M. 1994. Shoring up persistent applications. In *Proceedings of the ACM SIGMOD Conference on Management of Data* (Minneapolis, MI, May), 383–394.

CAREY, M. AND KOSSMANN, D. 1997. On saying "enough already!" in SQL. In *Proceedings of the ACM SIGMOD Conference on Management of Data* (Tucson, AZ, May), 219–230.

CAREY, M. AND KOSSMANN, D. 1998. Reducing the braking distance of an SQL query engine. In *Proceeding of the Conference on Very Large Data Bases (VLDB)* (New York, Aug.), 158–169.

CAREY, M. AND LU, H. 1986. Load balancing in a locally distributed database system. In *Proceedings of the ACM SIGMOD Conference on Management of Data* (Washington, DC, June), 108–119.

CATTELL, R., BARRY, D., BARTELS, D., BERLER, M., EASTMAN, J., GAMERMAN, S., JORDAN, D., SPRINGER, A., STRICKLAND, H., AND WADE, D. 1997. *The Object Database Standard: ODMG 2.0*. The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann Publishers, San Mateo, CA.

CERI, S. AND PELAGATTI, G. 1984. *Distributed Databases—Principles and Systems*. McGraw-Hill Inc., New York, San Francisco, Washington, D.C.

CHAMBERLIN, D., ASTRAHAN, M., KING, W., LORIE, R., MEHL, J., PRICE, T., SCHKOLNIK, M., SELINGER, P., SLUTZ, D., WADE, B., AND YOST, R. 1981. Support for repetitve transactions and ad hoc queries in System R. *ACM Transactions on Database Systems 6*, 1 (March), 70–94.

CHAUDHURI, S. AND GRAVANO, L. 1996. Optimizing queries over mulitmedia repositories. In *Proceedings of the ACM SIGMOD Conference on Management of Data* (Montreal, Canada, June), 91–102.

CHEN, C. AND ROUSSOPOULOS, N. 1994. Adaptive selectivity estimation using query feedback. In *Proceedings of the ACM SIGMOD Conference on Management of Data* (Minneapolis, MI, May), 161–172.

COLE, R. AND GRAEFE, G. 1994. Optimization of dynamic query evaluation plans. In *Proceedings of the ACM SIGMOD Conference on Management of Data* (Minneapolis, MI, May), 150–160.

COPELAND, G., ALEXANDER, W., BOUGHTER, E., AND KELLER, T. 1988. Data placement in bubba. In *Proceedings of the ACM SIGMOD Conference on Management of Data* (Chicago, IL, May), 99–108.

D'ANDREA, A. AND JANUS, P. 1996. UniSQL's next-generation object-relational database management system. *ACM SIGMOD Record 25*, 3 (Sept.), 70–76.

DAR, S., FRANKLIN, M., JÓNSSON, B., SRIVASTAVA, D., AND TAN, M. 1996. Semantic data caching and replacement. In *Proceedings of the Conference on Very Large Data Bases (VLDB)* (Bombay, India, Sept.), 330–341.

DAVIDSON, S., GARCIA-MOLINA, H., AND SKEEN, D. 1985. Consistency in partitioned networks. *ACM Computing Surveys 17*, 2 (Sept.), 341–370.

DAYAL, U. 1983. Processing queries over generalization hierarchies in a multidatabase system. In *Proceedings of the Conference on Very Large Data Bases (VLDB)* (Florence, Italy, Oct.), 342–353.

DESHPANDE, P., RAMASAMY, K., SHUKLA, A., AND NAUGHTON, J. 1998. Caching multidimensional queries using chunks. In *Proceedings of the ACM SIGMOD Conference on Management of Data* (Seattle, WA, June), 259–270.

DESSLOCH, S., HÄRDER, T., MATTOS, N., MITSCHANG, B., AND THOMAS, J. 1998. KRISYS: Modeling concepts, implementation techniques, and client/server issues. *The VLDB Journal 7*, 2 (April), 79–95.

DEWITT, D., FUTTERSACK, P., MAIER, D., AND VELEZ, F. 1990. A study of three alternative workstation server architectures for object-oriented database systems. In *Proceedings of the Conference on Very Large Data Bases (VLDB)* (Brisbane, Australia, Aug.), 107–121.

DEWITT, D. AND GRAY, J. 1992. Parallel database systems: The future of high performance database systems. *Communications of the ACM 35*, 6 (June), 85–98.

DEWITT, D., LIEUWEN, D., AND MEHTA, M. 1993. Parallel pointer-based join techniques for object-oriented databases. In *Proceedings of the International IEEE Conference on Parallel and Distributed Information Systems* (San Diego, CA, Jan.).

DOGAC, A., HALICI, U., KILIC, E., ÖZHAN, G., OZCAN, F., NURAL, S., DENGI, C., MANCUHAN, S., ARPINAR, B., KOKASL, P., AND EVRENDILEK, C. 1996. METU interoperable database system. In *Proceedings of the ACM SIGMOD Conference on Management of Data* (Montreal, Canada, June), 552.

DOPPELHAMMER, J., HÖPPLER, T., KEMPER, A., AND KOSSMANN, D. 1997. Database performance in the real world: TPC-D and SAP R/3. In *Proceedings of the ACM SIGMOD Conference on Management of Data* (Tucson, AZ, May), 123–134.

DU, W., KRISHNAMURTHY, R., AND SHAN, M.-C. 1992. Query optimization in heterogeneous DBMS. In *Proceedings of the Conference on Very Large Data Bases (VLDB)* (Vancouver, Canada, Aug.), 277–291.

DU, W., SHAN, M.-C., AND DAYAL, U. 1995. Reducing multidatabase query response time by tree balancing. In *Proceedings of the ACM SIGMOD Conference on Management of Data* (San Jose, CA, May), 293–303.

EICKLER, A., GERLHOF, C., AND KOSSMANN, D. 1995. A performance evaluation of OID mapping techniques. In *Proceedings of the Conference on Very Large Data Bases (VLDB)* (Zürich, Switzerland, Sept.), 18–29.

EICKLER, A., KEMPER, A., AND KOSSMANN, D. 1997. Finding data in the neighborhood. In *Proceedings of the Conference on Very Large Data Bases (VLDB)* (Athens, Greece, Aug.), 336–345.

EPSTEIN, R., STONEBRAKER, M., AND WONG, E. 1978. Query processing in a distributed relational database system. In *Proceedings of the ACM SIGMOD Conference on Management of Data* (Austin, TX, June), 169–180.

EVRENDILEK, C., DOGAC, A., NURAL, S., AND OZCAN, F. 1997. Multidatabase query optimization. *Distributed and Parallel Databases 5*, 1 (Jan.), 77–114.

FAGIN, R. 1996. Combining fuzzy information from multiple systems. In *Proceedings of the ACM SIGMOD/SIGACT Conference on Principle of Database Systems (PODS)* (Montreal, Canada, June), 216–226.

FAGIN, R. AND WIMMERS, E. 1997. Incorporating user preferences in multimedia queries. In *Proceedings of the International Conference on Database Theory (ICDT)*, Volume 1186 of *Lecture Notes in Computer Science (LNCS)* (Jan.), 247–261. Springer-Verlag.

FERGUSON, D., NIKOLAOU, C., SAIRAMESH, J., AND YEMINI, Y. 1996. Economic models for allocating resources in computer systems. In S. CLEARWATER ED., *Market based Control of Distributed Systems*. World Scientific Press.

FERGUSON, D., NIKOLAOU, C., AND YEMINI, Y. 1993. An economy for managing replicated data in autonomous decentralized systems. In *Proceedings International Symposium on Autonomous and Decentralized Systems* (Kawasaki, Japan).

FLORESCU, D., KOSSMANN, D., AND MANOLESCU, I. 2000. Integrating keyword search into XML query processing, In *Proceedings of the WWW Conference* (WWW9) (Amsterdam, The Netherlands, May).

FLORESCU, D., LEVY, A., AND MENDELZON, A. 1998. Database techniques for the worldwide web: A survey. *ACM SIGMOD Record 27*, 3 (Sept.), 59–74.

FRANKLIN, M., CAREY, M., AND LIVNY, M. 1993. Local disk caching for client-server database systems. In *Proceedings of the Conference on Very Large Data Bases (VLDB)* (Dublin, Ireland, Aug.), 543–554.

FRANKLIN, M., CAREY, M., AND LIVNY, M. 1997. Transactional client-server cache consistency: Alternatives and performance. *ACM Transactions on Database Systems 22*, 3 (Sept.), 315–363.

FRANKLIN, M., JÓNSSON, B., AND KOSSMANN, D. 1996. Performance tradeoffs for client-server query processing. In *Proceedings of the ACM SIGMOD Conference on Management of Data* (Montreal, Canada, June), 149–160.

FRANKLIN, M. AND ZDONIK, S. 1998. Data in your face: Push technology in perspective. In *Proceedings of the ACM SIGMOD Conference on Management of Data* (Seattle, Wash, June), 516–519.

GANGULY, S., GOEL, A., AND SILBERSCHATZ, A. 1996. Efficient and accurate cost models for parallel query optimizaton. In *Proceedings of the ACM SIGMOD/SIGACT Conference on Principles of Database Systems (PODS)* (Montreal, Canada, June), 172–181.

GANGULY, S., HASAN, W., AND KRISHNAMURTHY, R. 1992. Query optimization for parallel execution. In *Proceedings of the ACM SIGMOD Conference on Management of Data* (San Diego, CA, June), 9–18.

GARDARIN, G., GRUSER, J.-R., AND TANG, Z.-H. 1996. Cost-based selection of path expression processing algorithms in object-oriented databases. In *Proceedings of the Conference on Very Large Data Bases (VLDB)* (Bombay, India, Sept.), 390–401.

GRAEFE, G. 1990. Encapsulation of parallelism in the volcano query processing system. In *Proceedings of the ACM SIGMOD Conference on Management of Data* (Atlantic City, NJ, June), 102–111.

GRAEFE, G. 1993. Query evaluation techniques for large databases. *ACM Computing Surveys 25*, 2 (June), 73–170.

GRAEFE, G. 1995. The cascades framework for query optimization. *IEEE Data Engeneering Bulletin 18*, 3 (Sept.), 19–29.

GRAEFE, G. 1996. Iterators, schedulers, and distributed-memory parallelism. *Software Practice and Experience 26*, 4 (April), 427–452.

GRAEFE, G. AND DEWITT, D. 1987. The EXODUS optimizer generator. In *Proceedings of the ACM SIGMOD Conference on Management of Data* (San Francisco, CA, May), 160–172.

GRAEFE, G. AND MCKENNA, W. 1993. The Volcano optimizer generator: Extensibility and efficient search. In *Proceedings of the IEEE Conference on Data Engineering* (Vienna, Austria, April), 209–218.

GRAEFE, G. AND WARD, K. 1989. Dynamic query evaluation plans. In *Proceedings of the ACM SIGMOD Conference on Management of Data* (Portland, OR, May), 358–366.

GRAVANO, L., CHANG, C.-C., GARCIA-MOLINA, H., AND PAEPCKE, A. 1997. STARTS: stanford proposal for internet meta-searching. In *Proceedings of the ACM SIGMOD Conference on Management of Data* (Tucson, AZ, May), 207–218.

GRAVANO, L. AND GARCIA-MOLINA, H. 1997. Merging ranks from heterogeneous internet sources. In *Proceedings of the Conference on Very Large Data Bases (VLDB)* (Athens, Greece, Aug.), 196–205.

GRAY, J., BOSWORTH, A., LAYMAN, A., AND PIRAHESH, H. 1996. Data cube: a relational aggregation operator generalizing group-by, cross-tab, and subtotal. In *Proceedings of the IEEE Conference on Data Engineering* (New Orleans, LA, Feb.), 152–159.

GRAY, J. AND REUTER, A. 1993. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, San Mateo, CA.

GUPTA, A., HARINARAYAN, V., AND RAJARAMAN, A. 1997. Virtual data technology. *ACM SIGMOD Record 26*, 4 (Dec.), 57–61.

GWERTZMAN, J. AND SELTZER, M. 1994. The Case for Geographical Push-Caching. Technical Report HU TR-34-94, Harvard University, Cambridge, MA.

HAAS, L., FREYTAG, J. C., LOHMAN, G., AND PIRAHESH, H. 1989. Extensible query processing in starburst. In *Proceedings of the ACM SIGMOD Conference on Management of Data* (Portland, OR, USA, May), 377–388.

HAAS, L., KOSSMANN, D., WIMMERS, E., AND YANG, J. 1997. Optimizing queries across diverse data sources. In *Proceedings of the Conference on Very Large Data Bases (VLDB)* (Athens, Greece, Aug.), 276–285.

HAGMANN, R. AND FERRARI, D. 1986. Performance analysis of several back-end database architectures. *ACM Transactions on Database Systems 11*, 1 (March), 1–26.

HAMILTON, G., CATTELL, R., AND FISHER, M. 1997. *JDBC Database Access with Java: A Tutorial and Annotated Reference*. Addison-Wesley, Reading, MA.

HÄRDER, T., MITSCHANG, B., NINK, U., AND RITTER, N. 1995. Workstation/server-architekturen für datenbankbasierte ingenieuranwendungen. *Informatik—Forschung und Entwicklung 10*, 2 (May), 55–72.

HARINARAYAN, V., RAJARAMAN, A., AND ULLMAN, J. 1996. Implementing data cubes efficiently. In *Proceedings of the ACM SIGMOD Conference on Management of Data* (Montreal, Canada, June), 205–216.

HASAN, W. AND MOTWANI, R. 1995. Coloring away communication in parallel query optimization. In *Proceedings of the Conference on Very Large Data Bases (VLDB)* (Zürich, Switzerland, Sept.), 239–250.

HONG, W. AND STONEBRAKER, M. 1990. Parallel Query Processing in XPRS. Technical report UCB/ERL M90/47 (May), Department of Industrial Engineering and Operations Research and School of Business Administration, University of California, Berkeley, CA.

IEEE Data Engineering Bulletin. 1998. Special issue on interoperability. *IEEE Data Engineering Bulleting*, 21, 3.

IOANNIDIS, Y. AND KANG, Y. 1991. Left-deep vs. bushy trees: an analysis of strategy spaces and its implications for query optimization. In *Proceedings of the ACM SIGMOD Conference on Management of Data* (Denver, CO, May), 168–177.

IOANNIDIS, Y., NG, R., SHIM, K., AND SELLIS, T. 1992. Parametric query optimization. In *Proceedings of the Conference on Very Large Data Bases (VLDB)* (Vancouver, Canada, Aug.), 103–114.

IVES, Z., FLORESCU, D., FRIEDMAN, M., LEVY, A., AND WELD, D. 1999. An adaptive query execution engine for data integration. In *Proceedings of the ACM SIGMOD Conference on Management of Data* (Philadelphia, PA, USA, June), 299–310.

JENQ, B., WOELK, D., KIM, W., AND LEE, W. 1990. Query processing in distributed ORION. In *Proceedings of the International Conference on Extending Database Technology (EDBT)* (Venice, Italy, March), 169–187.

KABRA, N. AND DEWITT, D. 1998. Efficient mid-query re-optimization for sub-optimal query execution plans. In *Proceedings of the ACM SIGMOD Conference on Management of Data* (Seattle, WA, June), 106–117.

KELLER, A. AND BASU, J. 1994. A predicate-based caching scheme for client-server database architectures. In *Proceedings of the International IEEE Conference on Parallel and Distributed Information Systems* (Austin, TX, Sept.), 229–238.

KELLER, A., JENSEN, R., AND AGRAWAL, S. 1993. Persistence software: Bridging object-oriented programming and relational databases. In *Proceedings of the ACM SIGMOD Conference on Management of Data* (Washington, DC, May), 523–528.

KELLER, T., GRAEFE, G., AND MAIER, D. 1991. Efficient assembly of complex objects. In *Proceedings of the ACM SIGMOD Conference on Management of Data* (Denver, CO, May), 148–158.

KEMPER, A. AND KOSSMANN, D. 1994. Dual-buffering strategies in object bases. In *Proceedings of the Conference on Very Large Data Bases (VLDB)* (Santiago, Chile, Sept.), 427–438.

KEMPER, A., KOSSMANN, D., AND MATTHES, F. 1998. SAP R/3: A Database Application System. Tutorial handouts for the ACM SIGMOD Conference, Seattle, WA. http://www.db.fmi.uni-passau.de/publications/tutorials/.

KIM, W., GARZA, J., BALLOU, N., AND WOELK, D. 1990. Architecture of the ORION next-generation database system. *IEEE Transactions on Knowledge and Data Engineering 2*, 1 (March), 109–124.

KIMBALL, R. AND STREHLO, K. 1995. Why decision support fails and how to fix it. *ACM SIGMOD Record 24*, 3 (Sept.), 92–97.

KOSSMANN, D., FRANKLIN, M., AND DRASCH, G. 2000. Cache Investment: Integrating query optimization and dynamic data placement. *ACM Trans. Data Syst.*

KOSSMANN, D. AND STOCKER, K. 2000. Iterative dynamic programming: A new class of query optimization algorithms. *ACM Transactions on Database Systems 25*, 1 (March).

LAHIRI, T., JOSHI, A., JASUJA, A., AND CHATTERJEE, S. 1998. 50,000 users on an Oracle8 Universal Server database. In *Proceedings of the ACM SIGMOD Conference on Management of Data* (Seattle, WA, June), 528–530.

LANZELOTTE, R., VALDURIEZ, P., AND ZAIT, M. 1993. On the effectiveness of optimization search strategies for parallel execution spaces. In *Proceedings of the Conference on Very Large Data Bases (VLDB)* (Dublin, Ireland, Aug.), 493–504.

LEVY, A. 1999. Answering Queries Using Views: A Survey. In preparation.

LEVY, A., RAJARAMAN, A., AND ORDILLE, J. 1996. Querying heterogeneous information sources using source descriptions. In *Proceedings of the Conference on Very Large Data Bases (VLDB)* (Bombay, India, Sept.), 251–262.

LOHMAN, G. 1988. Grammar-like functional rules for representing query optimization alternatives. In *Proceedings of the ACM SIGMOD Conference on Management of Data* (Chicago, IL, May), 18–27.

LOMET, D. 1996. Replicated indexes for distributed data. In *Proceedings of the International IEEE Conference on Parallel and Distributed Information Systems* (Miami Beach, FL, Dec.).

LORIE, R. AND WADE, B. 1979. The Compilation of a High Level Data Language. Technical Report RJ 2598, IBM Research, San Jose, CA.

LU, H. AND CAREY, M. 1985. Some experimental results on distributed join algorithms in a local network. In *Proceedings of the Conference on Very Large Data Bases (VLDB)* (Stockholm, Sweden), 229–304.

LUOTONEN, A. AND ALTIS, K. 1994. World-Wide Web Proxies. Technical report (April), CERN, Geneva, Switzerland.

MACKERT, L. AND LOHMAN, G. 1986. R* optimizer validation and performance evaluation for distributed queries. In *Proceedings of the Conference on Very Large Data Bases (VLDB)* (Kyoto, Japan), 149–159.

MAIER, D., GRAEFE, G., SHAPIRO, L., DANIELS, S., KELLER, T., AND VANCE, B. 1994. Issues in distributed object assembly. In T. ÖZSU, U. DAYAL, AND P. VALDURIEZ EDS., *Distributed Object Management* (San Mateo, CA, May 1994), 165–181. Morgan Kaufmann Publishers. International Workshop on Distributed Object Management.

MCHUGH, J. AND WIDOM, J. 1999. Query optimization for XML. In *Proceedings of the Conference on Very Large Data Bases (VLDB)* (Edinburgh, GB, Sept.), 315–326.

MELTON, J. AND SIMON, A. 1993. *Understanding the New SQL: A Complete Guide*. Morgan Kaufmann Publishers, San Mateo, CA.

MISHRA, P. AND EICH, M. 1992. Join processing in relational databases. *ACM Computing Surveys 24*, 1 (March), 63–113.

NAACKE, H., GARDARIN, G., AND TOMASIC, A. 1998. Leveraging mediator cost models with heterogeneous data sources. In *Proceedings IEEE Conference on Data Engineering* (Orlando, FL).

O'TOOLE, J. AND SHRIRA, L. 1994. Opportunisic Log: Efficient Reads in a Reliable Object Server. Technical Report MIT/LCS-TM-506 (March), Massachusetts Institute of Technology, Cambridge, MA 02139.

OZCAN, F., NURAL, S., KOKSAL, P., EVRENDILEK, C., AND DOGAC, A. 1996. Dynamic query optimization on a distributed object management platform. In *Proceedings of the International Conference on Information and Knowledge Management* (Rockville, MD, Nov.), 117–124.

OZCAN, F., NURAL, S., KOKSAL, P., EVRENDILEK, C., AND DOGAC, A. 1997. Dynamic query optimization in multidatabases. *IEEE Data Engineering Bulletin 20*, 3 (Sept.), 38–45.

ÖZSU, T. AND VALDURIEZ, P. 1999. *Principles of Distributed Database Systems* (second ed.). Prentice Hall, Englewood Cliffs, NJ.

PAPAKONSTANTINOU, Y., GARCIA-MOLINA, H., AND WIDOM, J. 1995a. Object exchange across heterogeneous information sources. In *Proceedings of the IEEE Conference on Data Engineering* (Taipeh, Taiwan, 1995), 251–260.

PAPAKONSTANTINOU, Y., GUPTA, A., GARCIA-MOLINA, H., AND ULLMAN, J. 1995b. A query tranlation scheme for rapid implementation of wrappers. In *Proceedings of the Conference on Deductive and Object-Oriented Databases (DOOD)* (Dec.), 161–186.

PAPAKONSTANTINOU, Y., GUPTA, A., AND HAAS, L. 1996. Capabilities-based query rewriting in mediator systems. In *Proceedings of the International IEEE Conference on Parallel and Distributed Information Systems* (Miami Beach, FL, Dec.).

PIRAHESH, H., HELLERSTEIN, J., AND HASAN, W. 1992. Extensible/rule based query rewrite optimization in starburst. In *Proceedings of the ACM SIGMOD Conference on Management of Data* (San Diego, CA, June), 39–48.

QUASS, D. AND WIDOM, J. 1997. On-line warehouse view maintenance. In *Proceedings of the ACM SIGMOD Conference on Management of Data* (Tucson, AZ, May), 393–404.

RAMAKRISHNAN, R. 1997. *Database Management Systems*. McGraw-Hill, Inc., New York, San Francisco, Washington, DC.

RELLY, L., SCHULDT, H., AND SCHEK, H.-J. 1998. Exporting database functionality—the `concert` way. *IEEE Data Engeneering Bulletin 21*, 3 (Sept.), 40–48.

ROTH, M. T., OZCAN, F., AND HAAS, L. 1999. Cost models DO matter: Providing cost information for diverse data sources in a federated system. In *Proceedings of the Conference on Very Large Data Bases (VLDB)* (Edinburgh, GB, Sept.), 599–610.

ROTH, M. T. AND SCHWARZ, P. 1997. Don't scrap it, wrap it! A wrapper architecture for legacy data sources. In *Proceedings of the Conference on Very Large Data Bases (VLDB)* (Athens, Greece, Aug.), 266–275.

ROUSSOPOULOS, N. 1991. The incremental access method of view cache: Concepts, algorithms, and cost analysis. *ACM Transactions on Database Systems 16*, 3 (Sept.), 535–563.

ROUSSOPOULOS, N., CHEN, C., KELLEY, S., DELIS, A., AND PAPAKONSTANTINOU, Y. 1995. The adms project: views r us. *IEEE Data Engeneering Bulletin 18*, 2 (June), 19–28.

SCHEUERMANN, P., SHIM, J., AND VINGRALEK, R. 1996. WATCHMAN: a data warehouse intelligent cache manager. In *Proceedings of the Conference on Very Large Data Bases (VLDB)* (Bombay, India, Sept.), 51–62.

SCHNEIDER, D. AND DEWITT, D. 1990. Tradeoffs in processing complex join queries via hashing in multiprocessor database machines. In *Proceedings of the Conference on Very Large Data Bases (VLDB)* (Brisbane, Australia, Aug.), 469–480.

SELINGER, P., ASTRAHAN, M., CHAMBERLIN, D., LORIE, R., AND PRICE, T. 1979. Access path selection in a relational database management system. In *Proceedings of the ACM SIGMOD Conference on Management of Data* (Boston, MA, May), 23–34.

SELLIS, T. 1988. Multiple-query optimization. *ACM Transactions on Database Systems 13*, 1 (March), 23–52.

SHAN, M.-C., AHMED, R., DAVIS, J., DU, W., AND KENT, W. 1994. Pegasus: A heterogeneous information management system. In W. KIM ED., *Modern Database Systems*, Chapter 32. Reading, MA. ACM Press (Addison-Wesley publishers).

SHEKITA, E. AND CAREY, M. 1990. A performance evaluation of pointer-based joins. In *Proceedings of the ACM SIGMOD Conference on Management of Data* (Atlantic City, NJ, May), 300–311.

SHETH, A. AND LARSON, J. 1990. Federated database systems for managing distributed, heterogeneous, and autonmous databases. *ACM Computing Surveys 22*, 3 (Sept.), 183–236.

SIDELL, J., AOKI, P., BARR, S., SAH, A., STAELIN, C., STONEBRAKER, M., AND YU, A. 1996. Data replication in Mariposa. In *Proceedings IEEE Conference on Data Engineering* (New Orleans, LA, Feb.), 485–494.

SILBERSCHATZ, A., KORTH, H., AND SUDARSHAN, S. 1997. *Database System Concepts* (third ed.). McGraw-Hill, Inc., New York, San Francisco, Washington, DC.

SRINIVANSAN, V. AND CAREY, M. 1992. Compensation-based on-line query processing. In *Proceedings of the ACM SIGMOD Conference on Management of Data* (San Diego, CA, June), 331–340.

STEINBRUNN, M., MOERKOTTE, G., AND KEMPER, A. 1997. Heuristic and randomized optimization for the join ordering problem. *The VLDB Journal 6*, 3 (Aug.), 191–208.

STOCKER, K., KOSSMANN, D., BRAUMANDL, R., AND KEMPER, A. 2001. Integrating semijoin reducers into state-of-the-art query processors. In *Proceedings of the IEEE Conference on Data Engineering* (Heidelberg, Germany, April.). IEEE Computer Society Press, Los Angeles, Calif.

STONEBRAKER, M. 1985. The design and implementation of distributed INGRES. Reading, MA. Addison-Wesley.

STONEBRAKER, M. 1986. The case for shared nothing. *IEEE Data Engeneering Bulletin 9*, 1 (March), 4–9.

STONEBRAKER, M. 1994. *Readings in Database Systems* (second ed.). Morgan Kaufmann Publishers, San Mateo, CA.

STONEBRAKER, M., AOKI, P., LITWIN, W., PFEFFER, A., SAH, A., SIDELL, J., STAELIN, C., AND YU, A. 1996. Mariposa: a wide-area distributed database system. *The VLDB Journal 5*, 1 (Jan.), 48–63.

TANENBAUM, A. 1989. *Computer Networks*. Prentice Hall, Englewood Cliffs, NJ.

TANENBAUM, A. 1992. *Modern Operating Systems*. Prentice Hall, Englewood Cliffs, NJ.

THOMAS, J., GERBES, T., HÄRDER, T., AND MITSCHANG, B. 1995. Implementing dynamic code assembly for client-based query processing. In *Proceedings of the International Symposium for Advanced Applications, (DASFAA)* (Singapore, April), 264–272.

TOMASIC, A., RASCHID, L., AND VALDURIEZ, P. 1998. Scaling acccess to distributed heterogeneous data sources with DISCO. *IEEE Transactions on Knowledge and Data Engineering 10*, 5 (Oct.), 808–823.

ULLMAN, J. 1988. *Principles of Data and Knowledge-Base Systems*, Vol. I. Computer Science Press, Woodland Hills, CA.

URHAN, T. AND FRANKLIN, M. 1999. Xjoin: Getting Fast Answers from Slow and Bursty Networks. Technical report CS-TR-3994 (Feb.), University of Maryland, College Park.

URHAN, T., FRANKLIN, M., AND AMSALEG, L. 1998. Cost based query scrambling for initial delays. In *Proceedings of the ACM SIGMOD Conference on Management of Data* (Seattle, WA, June), 130–141.

VALDURIEZ, P. AND GARDARIN, G. 1984. Join and semijoin algorithms for a multiprocessor database machine. *ACM Transactions on Database Systems 9*, 1 (March), 133–161.

WIDOM, J. 1995. Research problems in data warehousing. In *Proceedings of the International Conference on Information and Knowledge Management* (Baltimore, MD, Nov.), 25–30.

WIEDERHOLD, G. 1993. Intelligent integration of information. In *Proceedings of the ACM SIGMOD Conference on Management of Data* (Washington, DC, May), 434–437.

WILLIAMS, R., DANIELS, D., HAAS, L., LAPIS, G., LINDSAY, B., NG, P., OBERMARCK, R., SELINGER, P., WALKER, A., WILMS, P., AND YOST, R. 1981. R*: An Overview of the Architecture. IBM Research, San Jose, CA, RJ3325. Reprinted in: M. Stonebraker (ed.), *Readings in Database Systems*, Morgan Kaufmann Publishers, 1994, 515–536.

WILSHUT, A. AND APERS, P. 1991. Dataflow query execution in a parallel main memory. In *Proceedings of the International IEEE Conference on Parallel and Distributed Information Systems* (Miami, FL, Dec.), 68–77.

WOLFSON, O., JAJODIA, S., AND HUANG, Y. 1997. An adaptive data replication algorithm. *ACM Transactions on Database Systems 22*, 42 (June), 255–314.

YANG, J., KARLAPALEM, K., AND LI, Q. 1997. Algorithms for materialized view design in data warehousing environment. In *Proceedings of the Conference on Very Large Data Bases (VLDB)* (Athens, Greece, Aug.), 136–145.

YU, C. AND CHANG, C. 1984. Distributed query processing. *ACM Computing Surveys 16*, 4 (Dec.), 399–433.

YU, C. AND MENG, W. 1997. *Principles of Database Query Processing for Advanced Applications*. Morgan Kaufmann Publishers, San Mateo, CA.

ZAHARIOUDAKIS, M. AND CAREY, M. 1997. Highly concurrent cache consistency for indices in client-server database systems. In *Proceedings of the ACM SIGMOD Conference on Management of Data* (Tucson, AZ, May), 50–61.

ZHU, Q. AND LARSON, P. 1994. A query sampling method of estimating local cost parameters in a multidatabase system. In *Proceedings IEEE Conference on Data Engineering* (Houston, TX, USA, Feb.), 144–153.

Received July 1998; revised May 1999; accepted November 1999