The State of the Art in Distributed Query Processing

Donald Kossmann
University of Passau
94030 Passau, Germany
http://www.db.fmi.uni-passau.de/~kossmann

Abstract

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., network technology as manifested by the Internet), there are a number of issues which 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) 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; (3) security and autonomy considerations set up by various different administrators of individual sites of the system need to be taken into account.

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 intra-query 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, multi-tier, 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; multi-tier 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 seventies. At that time, the main focus was to support distributed data management

for large corporations and organizations that keep their data at different offices or subsidiaries. Although there was a clear need and many good ideas and prototypes (e.g., System R* [WDH+81], SDD-1 [BGW+81], and Distributed Ingres [Sto85]), the early efforts in building distributed database systems never turned into commercial success [Sto94]. In some sense, 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; two, 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, in the late nineties, the situation has changed dramatically. Distributed data processing is becoming a commodity, and businesses are moving more and more towards distributed systems to manage and process their data. 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). The WWW is another arena that sparks a large number of new applications for distributed database and information systems. Of course, one reason for this breakthrough of distributed data processing are recent technical achievements (hardware, software protocols, standards, etc.) which have made large-scale distributed computing possible. A second reason for this breakthrough are changed business requirements and new opportunities that make 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 a 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 processors 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 one software package can meet all the requirements of a company. Companies must, therefore, install several different packages, each potentially with their own database, and the result is a distributed database system. Even single software packages 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 stay competitive. To

give an example, people will (probably) not eat more Pizza because of the Internet, but a Pizza delivery service is definitely going to lose market share in the long run if it does not allow people to order Pizza on the web.

So, many different flavors of distributed systems exist, and 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. That is, we assume that users and application programs issue queries using a declarative query language like SQL [MS93] or OQL [CBB+97] and without knowing where and in which format the data is stored in the distributed system, and our goal is to evaluate such queries as efficiently as possible in order to minimize the time that users must wait for answers or 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: we will, for example, describe the design of a query optimizer that compiles a query for execution and determines the best possible way among many ways to execute a query, and we will show how techniques such as caching and replication can be used to improve the performance of queries in a distributed environment. We will also cover specific query processing techniques for client-server, multi-tier, and heterogeneous database and information systems which represent the most important classes of distributed database systems found in practice.

1.2 Scope of this Paper and Related Surveys

Obviously, a very large body of work in the general area of database systems exists. All this work can roughly be classified into work on architectures and techniques for transaction processing (i.e., quickly processing small update operations for, say, order line item processing), 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. Of course, we will not even be able to cover just a small fraction of all of this work. We will completely concentrate on query processing and mostly focus on the relational model and SQL-style queries, and note that work on transaction processing techniques as well as work on special-purpose techniques for query processing in object-oriented and deductive database systems is orthogonal to the work surveyed in this paper. Transaction processing has, for example, been thoroughly investigated in [GR93], and there exists a number of books that explain object-oriented and deductive database systems; e.g., [ZM90, KM94, Kim94] for object-oriented or [Ull88] for deductive databases. Also, we will assume that the reader has basic knowledge about database systems and is familiar with SQL and the relational data model. Good introductory textbooks are [Dat95, SKS97, Ram97].

One particular kind of distributed system are parallel database systems. Distributed and parallel database systems share several properties and goals—in particular if we think about so-called "shared-nothing" parallel systems that are composed of a set of nodes that have their own processor, main memory and disk(s) [Sto86]. Parallel database systems are an alternative to centralized database systems with the purpose to improve transaction and query response times and the availability of the system. Parallel systems, therefore, emphasize the cost/scalability arguments described above, while the distributed systems we are going to deal with in this paper often fight, in addition, with problems such as

heterogeneity of its components. While some query processing techniques are useful for both kinds of systems, researchers in both areas have developed a set of 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 we 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.) in any detail here. An excellent overview on parallel database systems is given in [DG92].

In terms of related work, there have been a couple of surveys on distributed query processing; e.g., a paper by Yu and Chang [YC84] and parts of the books by Ceri and Pelagatti [CP84], Özsu and Valduriez [ÖV91], and Yu and Meng [YM97] are devoted to distributed query processing. These surveys, however, are mostly focussed on the presentation of the techniques used in the early prototypes of the seventies and eighties. 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 couple of 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 which are useful for any kind of distributed database system;
- Section 3 looks closer at query processing in one particular and very important class of distributed database systems: client-server and multi-tier database systems;
- Section 4 deals with the query processing issues that arise in heterogeneous database systems; i.e., systems that are composed of several (autonomous) component databases with different schemas, varying query processing capabilities, and 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 overall, globally 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 query execution 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

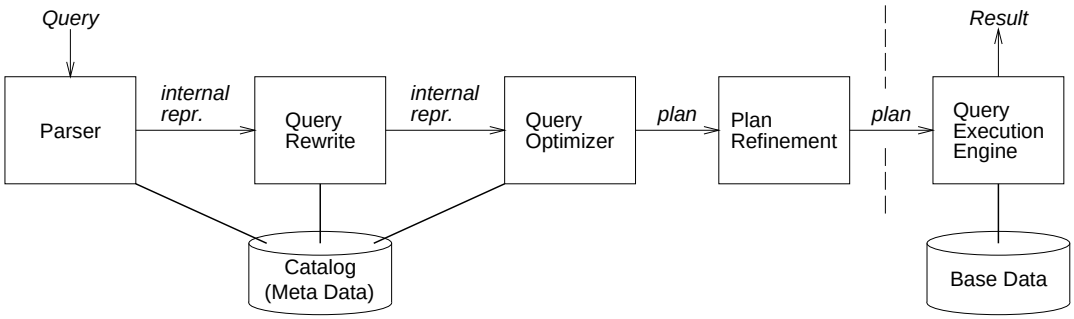


Figure 1: Phases of Query Processing [HFLP89]

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 then discuss which techniques 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 developed as part of IBM's Starburst project [HFLP89], and it can be used for any kind of database system including centralized, distributed, or parallel systems. The query processor gets an (SQL or OQL) query as input, translates this query in several phases into an executable query plan, and then executes the plan in order to find the results of the query. In the first phase, the query is parsed, checked for syntactic errors, and translated into an *internal representation* that can easily be processed by the following phases. Parsing involves checking the existence of tables and other objects used in a query and, therefore, involves access to the *global* schema of the distributed database which is stored in the *catalog*. In all, the development of *parsers* is very well-understood [ASU87], and the development of an SQL or OQL parser for a database system is no worse than the development of a parser for programming languages like C++, and tools like *flex* and *bison* can be used. Furthermore, it should be noted that the same parser can be used for a centralized as for a distributed database system because one of the duties of the query processor is to provide transparent access so that the user need not even know whether he or she deals with a centralized or with a distributed database system.

The purpose of the next three phases is to find the best or lowest-cost strategy to execute a query taking into account that there are very many (actually indefinitely many) different ways to execute a query and get the right results. *Query rewrite* carries out optimizations which are good regardless of the *physical* state of the system (e.g., the size of tables, presence of indexes, locations of copies of tables, speed of machines, etc.). To do so, query rewrite will consider, for example, integrity constraints of the database and information about how the tables of the global schema of the distributed system are partitioned; integrity constraints and the partitioning schema are also stored in the catalog. The *query optimizer* carries out optimizations that do depend on the *physical* state of the system which is also described, as well as possible, in the catalog. To this end, the query optimizer enumerates alternative ways to execute a query and chooses the best one using a cost model. *Plan refinement* carries out another set of simple trans-

formations that eliminate some of the wrinkles introduced by the query optimizer; this way, the implementation of the optimizer can be simplified. 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 so-called *canned* query which is part of an application program (embedded SQL), then the plan is stored in the database and executed by the query execution engine every time the application program is executed [CAK+81]. We will describe query rewrite, query optimization, catalog management, and query execution in more detail in the following subsections.

It should be noted that the (Starburst) architecture shown in Figure 1 is not the only possible way to process queries, and 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 [GD87, GM93, Gra95] and is used in several commercial database products (e.g., Microsoft's latest SQL Server product). In that architecture, query rewrite and query optimization are carried out in one phase and different query optimization algorithms are used (see Section 2.3.3). We concentrate on the Starburst approach shown in Figure 1 because it is a very popular approach, and it integrates very well all of the techniques that we will cover in this paper.

2.2 Query Rewrite

As stated above, the purpose of query rewrite is to transform a query in such a way that the query can be executed more efficiently under all circumstances—regardless of the size of tables, location of copies of tables or partitions of tables, hardware of sites, etc. Specifically, the query rewrite component will eliminate redundant predicates, add predicates if they help reducing the size of intermediate results or provide more flexibility to process joins, simplify expressions, push-down predicates, and unnest views and subqueries. In some situations, query rewrite can even deduce that the result of a query is empty so that the query need not be executed at all. To transform queries correctly, the query rewrite component considers properties of the Boolean algebra, SQL semantics, and integrity constraints of the database. As a simple example, consider the following transformation for a query that asks for `Squares` with a low area:

```
select *                                        select *
from Squares s                          ⟶      from Squares s
where s.length * s.length < 10,000              where s.length < 100
```

This transformation is possible by applying simple math and if there is an integrity constraint in the database that says that no `Square` has `length < 0`. Under all circumstances, the transformed query is cheaper to execute than the original query because only a simple predicate without any multiplications needs to be applied to every `Square` in the transformed query. In situations in which an index on `Squares.length` exists, the transformed query can be executed significantly cheaper than the original query because the `Squares.length` index can be used to execute the transformed query whereas that index cannot be used to execute the original query.

Many examples can be found that demonstrate the usefulness of a query rewrite component (see, e.g., [PHH92, PLH97, SHP+96]). One may argue that query rewrite is not necessary if programmers try to implement queries carefully; however, many queries are generated by application systems (e.g., GUI tools) that just try to get the query right

rather than finding an efficient formulation for a query, and even experienced programmers do not always write a query in the best possible way and use, for example, views or nested queries where this is not necessary.

One particular role that query rewrite plays in a distributed system and that it does not play in a centralized system is to transform queries that involve horizontally and vertically partitioned tables [CP84, ÖV91]. Consider, for example, the query from above that asks for `Squares` with `length < 100`, and assume that the `Squares` table is horizontally partitioned into three partitions: assume, for instance, that the first partition, S_1, contains all `Squares` with `length < 80`, S_2 contains all `Squares` with `80` $\leq$ `length < 160`, and S_3 contains all `Squares` with `length` $\geq$ `160`. A user would write the query in the same way as above using the *global* `Squares` table and not knowing that it is partitioned. Query rewrite would then transform the user's query into a *union* query and find out that only S_1 and S_2 need to be considered in order to answer the query and that the predicate `length < 100` need not be applied to the tuples of S_1. Note that this kind of rewrite makes sense regardless of where the partitions are stored, and in fact, query rewrite does not even care about the location of copies of partitions because this is a *physical* property of the database.

Query rewrite can be implemented by a set of transformation rules and a rule engine that iteratively applies these rules to a query. The transformation rules can be coded in C++ or in a special-purpose rule language. The mechanism of the rule engine is not trivial because often only the application of several rules in the right order results in the best transformation of a query and because it is, of course, important to transform a query very quickly. Specific issues in the design of such rule engines have been described in [PLH97], and we will not discuss them in any detail in this paper. Another important concern is to design the query rewrite component in an extensible way so that new rules like the *horizontal partitioning rules* from above can easily be added to the system without changing other rules or the rule engine. Furthermore, it is important to have appropriate data structures for the internal representation of queries making it easier to write clean transformation rules; the query rewrite work of [PHH92, PLH97] proposes the use of a special query graph model (QGM) as an internal representation of queries.

2.3 Query Optimization

We now turn to the *query optimizer* whose purpose it is to enumerate alternative plans which all might be good or bad depending on the state of the database and characteristics of the query and which chooses the best plan according to a cost model. Basically, a query optimizer can be characterized by its search space of plans, its cost model, and its enumeration algorithm. In the following, we will discuss basic options for all these three components; we will describe other and more elaborate options for query optimization in a distributed system when we talk about client-server and heterogeneous database systems in Sections 3 and 4.

2.3.1 Search Space

Before we start thinking about the implementation and the details of distributed query optimization, we need to get a picture of all the possible alternative ways to execute a query and how plans are represented; i.e., the *search space* of plans that are considered

when the optimizer tries to find the best plan for a query. In a centralized database system, the following options to execute a query exist:

1. **Access path selection.** The optimizer must consider plans that use an index to read a table as well as plans that read a whole table without an index. Indexes (e.g., B^+ trees [Com79]) are useful to apply, say, range predicates and/or to produce the tuples in the right order. Depending on the selectivity of the predicates (i.e., the number of tuples that qualify), however, indexes are not always useful for any kind of query; in particular, unclustered (secondary) indexes should be used with great care because their usage can involve a great deal of random disk IO.

2. **Join and aggregation methods.** There are many different ways to carry out a join or a group-by operation. Typically, these methods are based on sorting, hashing, or the use of indexes. The optimizer must find the right methods for every join and group-by operation of a given query depending on the size of the tables involved in the query and the selectivity of the query predicates.

3. **Join ordering.** Exploiting commutativity and associativity, a query involving Tables A, B, and C can, for example, be executed in the following orders: $(A \bowtie B) \bowtie C$, $C \bowtie (A \bowtie B)$, $(A \bowtie C) \bowtie B$, etc. The differences in cost can be substantial depending, again, on the size of the tables and the selectivity of the query predicates [IC91]. In addition, it is sometimes possible to order group-by and join operations in various different ways [YL94, CS94].

4. **Pipelining or materializing intermediate results.** Some operations like chains of nested-loop joins can be carried out in a pipelined fashion; that is, a result tuple produced by one operation is immediately consumed by the next operation and several operations run concurrently in the pipeline. An alternative is to materialize all the results produced by an operation and start the next operation only after all results have been produced so that the two operators do not run concurrently. Compared to pipelining, this materialization approach incurs additional cost for materializing (i.e., writing to disk) the intermediate results; on the positive side, the materialization approach is attractive if resources like main memory are scarce so that it is best to execute one operator at a time.

In addition to all these considerations, the optimizer of a distributed system must make the following decisions:

1. **Site selection.** The optimizer must decide at which site every operation of a query is to be executed. (Recall that in an ideal distributed system, every operation can be carried out at every site.)

2. **Use of special query execution techniques.** As we will see at the end of this section and in the following sections, there are a large number of special techniques that can be used in distributed databases to execute queries and that do not make sense in centralized database systems. At the end of this section, we will, for example, discuss various ways to ship data between sites, single-threaded vs. multi-threaded query execution, semi-join programs to execute inter-site join queries, special ways to process queries in the presence of horizontally partitioned data, and alternative

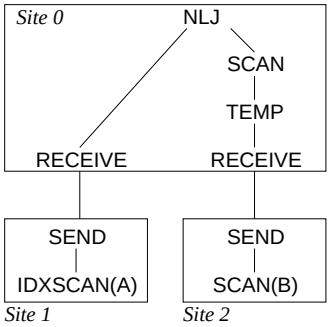


Figure 2: Example Query Evaluation Plan

ways to execute so-called *top N* queries and queries that involve path expressions. The optimizer must decide if any of these techniques are useful for a given query and, if so, in which way to make use of them.

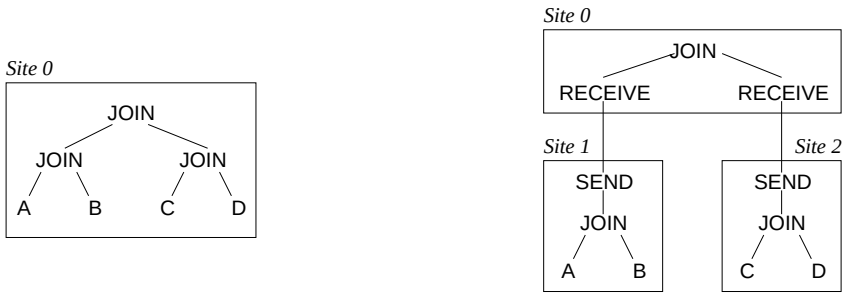
It is very important to notice that an optimizer can, in general, not make any decisions concerning one dimension independent of the other dimensions; the optimizer must, for example, consider all combinations of alternative site selections, join orderings, and join methods in order to find good plans. We will study this phenomenon in more detail in Section 3.3.3 when we study so-called two-step query optimization.

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 edges of a plan represent consumer-producer relationships of operators. Just to give an example, Figure 2 shows a 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).

2.3.2 Costing of 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 [ML86]. In this model, the cost of a plan is defined to be the total resource consumption of the plan. In a centralized system, the cost of an operator is composed of CPU costs plus disk IO costs, and the disk IO costs, in turn, are composed of seek, latency, and transfer costs. In a distributed system, communication costs of *send* and *receive* operators must also be considered and these communication 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. If, for example, a machine is known to be heavily loaded, then CPU cycles and disk IO operations can be modeled to be very expensive at that machine so that the optimizer will favor plans that carry out operators at cheaper and less loaded machines. Likewise, it can be modeled that it is more expensive to ship data from Passau (Germany) to Washington (USA) than from Passau to Munich (Germany).

In general, the more detailed the cost model, the more accurate are the cost estimates



Minimum Resource Consumption *Minimum Response Time*

Figure 3: Example Plans: Total Resource Consumption vs. Response Time

and the better are the plans generated by the query optimizer (i.e., with a good cost model, the optimizer will not select any bad plans believing they are good plans). Of course, a cost model can never be completely accurate because distributed systems and database operations are simply too complex and because the selectivity of predicates and the size of intermediate query results which are parameters of operator cost models cannot always be predicted accurately. However, there exists a large body of reasonably accurate cost models for various different operators and system configurations in the literature (e.g., [ML89, HR96, HCLS97]), and there is also a large body of work on histogram and sampling techniques in order to estimate the selectivity of predicates and the size of intermediate results of a query (e.g., [HS92, PIHS96, PI97]).

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 little resources as possible and avoid heavily loaded machines, then as many queries as possible can be executed at the same time. The classic cost model, however, does not consider intra-query parallelism and, therefore, 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 cheap.

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 3. Assuming that the cost of join processing are the same at all three sites and that copies of all tables are stored at all 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.

So, to find the plan with the lowest response time for a query (i.e., the second plan of Figure 3), we need to carry out query optimization with a cost model that estimates response time rather than total resource consumption, and such a cost model was devised in [GHK92]. 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 3, and these two joins and the top-level join can be carried out in a pipelined parallel fashion. Described on 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; the usage of the network, for example, is computed taking the bandwidth of the network and the volume of data transmitted to carry out all the operators that run in parallel into account. The response time of an entire group of operators that run in parallel is then computed using the *maximum* of the total resource consumption of the individual operators and the *maximum* of the total usage of all the shared resources. To go back to our example, this cost model would identify that the first plan has higher response time than the second plan because its response time is dominated by the high usage of the CPU and disks of Site 0 whereas the response time of the second plan is given by the maximum of the usage of the CPUs and disks of all three sites.

It should be noted that this cost model captures the effects of operator parallelism in a coarse-grained way, and looking closer at the model, it is possible to find situations in which inaccuracies of the cost model make the optimizer pick 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 (which is often true), and it has already been successfully used for query optimization in several studies (e.g., [FJK96, UFA98]). Like the classic cost model, it is able to evaluate a plan very quickly, and this is an important property because query optimization often involves applying the cost model to thousands of plans.

2.3.3 Enumeration Algorithms

Dynamic Programming Having defined the search space and ways to estimate the quality (cost or response time) of a plan, we are now ready to describe enumeration algorithms that try to find the best (i.e., cheapest or fastest) plan for a query as quickly as possible. A large number of alternative enumeration algorithms have been proposed in the literature (see, e.g., [SMK97] for an overview), but almost all commercial database systems rely on the same algorithm which is based on dynamic programming and which was developed as part of the System R project [SAC+79]. Figure 4 shows the basic dynamic programming algorithm for query optimization.[1] The algorithm works in a bottom-up way by building more complex plans from simpler plans. In the first step, the algorithm builds a so-called access-plan for every table involved in a query (Lines 1 and 2 of Figure 4). Then, the algorithm enumerates all two-way join plans using the access plans as building blocks. After that the algorithm builds three-way join plans, using access plans and two-way join plans as building blocks, and in this way the algorithm continues until it has enumerated n-way join plans which are complete plans for the query, if the query involves n tables.

In the literature, there has been a great deal of discussion concerning bushy or (left-) deep join plan enumeration [IK91, SD90, LVZ93]. 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 3 are bushy). The algorithm shown in Figure 4 enumerates all bushy plans, and considering all bushy plans is also the approach taken in most commercial database systems because often the best plan to execute a query is bushy and not deep; in particular in distributed systems [FJK96].

[1] Again, we concentrate on Starburst's variant of dynamic programming. A different, top-down dynamic-programming approach for query optimization has been presented in [GM93].

```
Input:  Query q involving Relations R_1, ..., R_n
Output: A plan to execute q
1:  for i := 1 to n do
2:      optPlan({R_i}) := accessPlan(R_i)
3:  for i := 2 to n do {
4:      for all S ⊆ {R_1, ..., R_n} such that |S| = i do {
5:          optPlan(S) := dummy plan with ∞ cost
6:          for all O, I such that S = O ∪ I and O ∩ I = ∅ do {
7:              p := joinPlan(optPlan(O), optPlan(I))
8:              if p.cost < optPlan(S).cost then
9:                  optPlan(S) := p
10:         }
11:     }
12: }
13: return optPlan({R_1, ..., R_n})
```

Figure 4: Dynamic Programming Algorithm

The beauty of the dynamic programming algorithm is that it discards (i.e., prunes) inferior building blocks early. To illustrate pruning, let us assume for the moment that the optimizer uses the classic cost model that estimates the total resource consumption of a plan or of a sub-plan. While enumerating two-way join plans, for example, the algorithm would consider an $A \bowtie B$ plan and a $B \bowtie A$ plan, but only the cheaper of the two plans would be recorded in the *optPlan({A,B})* structure[2] so that only the cheaper of the two would be considered as a building block for three-way, four-way, etc. join plans (using methods like hybrid hashing and nested-loops the cost of $A \bowtie B$ and $B \bowtie A$ may, in fact, be very different [Sha86]). As a result of this pruning, dynamic programming still finds the cheapest plan (according to the classic cost model), but dynamic programming does not consider all possible plans to execute a query and is, therefore, significantly faster than a naive exhaustive search. If the optimizer uses the response time model described in the previous section, pruning becomes slightly more complicated: in this case, dynamic programming must consider the resource consumption of every individual resource (i.e., network, CPUs and disks of all sites) in addition to the response times of sub-plans; details of this adjustment can be found in [GHK92].

Another advantage of the dynamic programming algorithm is that it is extensible [Loh88]. Figure 4 shows how join ordering which is only one dimension of the search space is carried out. The algorithm can be extended to enumerate alternative plans considering the other dimensions by extending the *accessPlan* and *joinPlan* functions. The *accessPlan* function could, for example, be extended so that it enumerates all possible index plans for a table, and likewise the *joinPlan* function could be extended to enumerate plans with all different join methods and with pipelined and materialized execution. Site selection can be carried out by extending both the *accessPlan* and the *joinPlan* functions: if, for example, Table A is replicated at Sites 1 and 2, then the *accessPlan* function would generate different access plans to read Table A from Sites 1 and 2. When extending a query optimizer to carry out site selection and other decisions, however, we need to be careful

[2]The *optPlan* data structure is sometimes called dynamic programming table.

which plans may be pruned, and dynamic programming typically needs to keep more than one plan in every *optPlan* entry. To carry out site selection in the presence of replication, for example, dynamic programming should keep all access plans for a table because the choice for the right access plan (i.e., which replica to use) can only be made after finding the right places to carry out the joins.

Greedy Enumeration Algorithms While the advantages of dynamic programming have driven most vendors to use this algorithm for query optimization, dynamic programming has one severe disadvantage: dynamic programming has exponential time and space complexity. Given today's hardware, dynamic programming can only be applied to queries involving fifteen or less tables in a centralized database system. In a distributed system with its significantly larger search space, the situation is even worse so that dynamic programming sometimes has trouble to optimize queries with eight or even fewer tables [SK98]. So, the challenge is to find an extensible enumeration algorithm that has low complexity and finds reasonably good plans. We cannot hope to be perfect here because query optimization was shown to be $\mathcal{NP}$ hard [IK84, SM97] so that the goal must be to find as good as possible plans with acceptable effort.

As stated above, a large class of algorithms have already been devised and [SMK97] contains a nice survey of many of these algorithms. Among all the candidate algorithms, however, probably the most promising class of algorithms are *greedy* algorithms. The first greedy algorithm for query optimization was proposed in the seventies [Pal74], and since then several other greedy algorithms have been devised (see, e.g., [SYT93, SK98]). Greedy enumeration algorithms have polynomial time and space complexity (typically, $\mathcal{O}(n * \log n)$ or $\mathcal{O}(n^2)$), and they are just as extensible as dynamic programming. In fact, most of the code of an existing dynamic programming-based optimizer can be reused to build a greedy variant of that optimizer (i.e., the *accessPlan* and *joinPlan* functions, all data structures, the pruning logic, etc.), and this is probably one of the most compelling reasons why system builders should consider greedy enumeration algorithms. Just like dynamic programming, greedy starts to generate access plans for all tables involved in a query and then generates and prunes two-way join plans based on these access plans. Other than dynamic programming, however, greedy only keeps the two-way join plans of one entry of the *optPlan* structure based on a selection criterion; for example, greedy might select the *optPlan({B,D})* plans and throw away the *optPlan({A,B}), optPlan({A,C})*, etc. plans after enumerating all two-way join plans, and the selection criterion in this case might be that *optPlan({B,D})* contains the cheapest of all two-way join plans or that the result of $B \bowtie D$ is smaller than the result of any other two-way join. After that, greedy considers $B \bowtie D$ as one (temporary) Table $\mathcal{T}$ and the plans stored in *optPlan({B,D})* as access plans for $\mathcal{T}$; that is, greedy turns the original optimization problem with n tables into an optimization problem with $n-1$ tables. Greedy continues to generate all two-way join plans with these $n-1$ tables and to select another *optPlan* entry and to reduce the optimization problem by another table with every step until only one (temporary) table is left. At this point, the algorithm terminates because it has generated a complete plan for the whole query.

This basic greedy algorithm is very fast, but it does not always generate good plans: in many cases, the plans are orders of magnitude more expensive than the best plan produced by dynamic programming [SMK97, SK98]. Better quality plans can be found with reasonable effort by two kinds of extensions of the basic greedy algorithm which are

incidentally both called iterative dynamic programming. The idea of the first extension is to select, say, four-way join plans rather than two-way join plans in every step of the algorithm; this way, the greedy algorithm is able to make better decisions by looking at bigger building blocks. The second extension is based on (re-)applying dynamic programming to certain parts of a query plan. Both extensions are described and thoroughly evaluated in [SK98].

2.4 Catalog Management

As shown in Figure 1, a catalog stores all the information needed in order to parse, rewrite, and optimize a query. It maintains the *global 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 which global tables have been partitioned and how they can be reconstructed), and *physical information* like the location of copies of partitions of tables, the presence of indexes, size of tables, and histograms that describe the value distribution of certain attributes.

In a centralized database system, catalog management can be carried out in a straightforward way, if the right data structures have been defined. In a distributed database system, the question arises *where* to store the catalog. The simplest approach is to have one (central) site take care of all catalog management. This approach works fairly well in local-area networks because catalog management is typically a very light-weight duty (significantly lighter-weight than, say, join processing) so that a centralized catalog manager is not likely to become the bottleneck of the system. In wide-area networks, it makes sense to replicate the catalog at several sites in order to reduce communication costs when sites all around the network require catalog information. It is also possible to cache catalog information at sites in order to reduce communication costs and avoid interaction with catalog managers [WDH+81]. Both replication and caching of catalog data are very effective in most systems because most catalogs are quite small (hundreds of KBs rather than GBs) and catalog information is rarely updated in most environments. In certain systems, however, the catalog can become very large and be frequently updated. In such systems, it makes sense to partition the catalog data and store catalog data where it is 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. Borrowing ideas from the implementation of the *domain name service*, such catalogs can be implemented in a hierarchical way as described in [EKK97].

2.5 Query Execution Techniques

We will now describe how queries are executed in a distributed database system. We will first describe how a query execution engine is structured using an *iterator model* for query execution, and then we will describe a series of specific query execution techniques for distributed databases. We will not describe any "standard" execution techniques for, say, joins and group-bys (e.g., hash-based algorithms) which can be used in centralized just as well as in distributed database systems and that have been described in other surveys [ME92, Gra93]. Instead, we will focus on query execution techniques which are particularly useful for distributed database systems.

2.5.1 The Iterator Model of Query Execution

The query execution engine provides generic implementations for every operator. In an object-oriented implementation (e.g., C++ or Java), for example, the query execution engine defines classes for nested-loop joins, hash joins, index scans, table scans, sort, send and receive operators, etc. To execute a plan, every operator of the plan is translated into an *iterator* which is simply an instance of the corresponding operator class, and the iterators are plugged together as specified by the consumer-producer relationships (i.e., edges) of the plan. Any two iterators can be plugged together, and thus, any plan produced by the optimizer can be implemented this way because every iterator has the same interface regardless of whether the iterator implements a join, sort, scan, etc. [Gra93]. Every iterator provides an *open* method which initializes the iterator, reserves memory and disk space and other resources needed to execute the iterator, and which calls the *open* method of its child iterators so that the child iterators are ready to produce tuples, too.[3] Furthermore, every iterator provides a *next* method which returns the iterator's next result tuple thereby—normally—calling the *next* method of its child iterators. Finally, every iterator provides a *close* method which releases all the resources allocated during *open* and which normally also calls the *close* method of its child iterators.

So, the general idea is that every iterator consumes tuples from its child iterators one at a time and returns result tuples one at a time to its parent iterator. The overall query results are produced (one by one) by the top-level iterator of the plan. It should be noted that some iterators consume all their input in their *open* phase before starting to produce tuples; examples are iterators like *sort*, *temp*, or iterators for certain join methods. Such a behavior is perfectly okay and does not break the basic *open-next-close* paradigm of iterators: just like any other iterator, *sort* provides *open, next*, and *close* methods which encapsulate how and when *sort* consumes tuples from its child iterator. Furthermore, note that some iterators consume input tuples in batches; that is they consume a block of tuples from their child iterators before returning result tuples one at a time. Again, such an approach does not break the basic *open-next-close* paradigm, and we will describe an example for which such blocking is useful in the next subsection.

2.5.2 Row Blocking

The first operators whose implementation we would like to describe are the *send* and *receive* operators. We already met those operators in the example plan of Figure 2 (Page 9). Their purpose is to ship data from one site to another site so that other operators for joins, sorts, group-bys, etc. can be implemented just as in a centralized database system without knowing that the data is located or produced at different sites. To make the shipping of data efficient, the *send* and *receive* operators work in a blockwise fashion. That is, rather than sending a tuple at a time, a *send* iterator consumes several tuples of its child iterator and sends these tuples as a batch. This approach is called row blocking, and it is employed by almost all commercial database systems today (e.g., IBM DB2, Oracle, etc.). This approach is much cheaper than the naive approach of sending a tuple at a time because the data is packed in significantly less messages. The size of the blocks is a parameter of the *send* and *receive* iterators, and this parameter is either set manually

[3] In fact, the *open* method of child iterators should be called first and resources should be allocated as late as possible [Gra93].

by a system administrator or automatically by a *scheduler* at running time[4]: the bigger the blocks, the less messages are required to ship a table or intermediate results; on the negative side, the bigger the blocks, the more memory is needed for the *send* and *receive* iterators and the less memory is left for other iterators.

Another 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, then a short delay in the network would immediately stop the execution of the parent iterator of the *receive* (i.e., the NLJ in the plan of Figure 2) because that operator would have no tuples to consume during this delay. Due to row blocking, the *receive* iterator has a reservoir of tuples and can feed its parent iterator even if the next block of tuples is delayed.

2.5.3 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; e.g., it is possible that the same data must be sent from Washington to Munich and Passau. If the network does not provide cheap ways to implement multicasts, it is sometimes a good idea to send the data from Washington to Munich and then from Munich to Passau rather than sending the data from Washington across the Atlantic twice. This technique has, for example, been implemented in the ORION-2 system [JWKL90].

Sometimes, this technique is even good in a (homogeneous) local area 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 very heavily loaded or has a slow CPU, then it might again be better if Passau gets the data from Munich rather than from Washington, whereas if Munich is heavily loaded and/or has a slow CPU, then Passau ought to get the data from Washington. Obviously, another option is that Passau gets the data from Washington and that Munich gets the data from Passau. The right choice must be made by the query optimizer.

2.5.4 Multi-Threaded Query Execution

As stated in Section 2.3.2, one of the goals of distributed query processing might be to exploit intra-query parallelism in order to minimize the *response time* of a query. In order to take the best advantage of intra-query parallelism, we sometimes need to establish several threads at a site [Gra90]. As an example, consider the plan of Figure 5 which implements the query $A_1 \cup A_2 \cup A_3$ at Site 0; 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, the

[4] Schedulers for query processors have been proposed in, e.g., [FNS91, YC91, DG95].

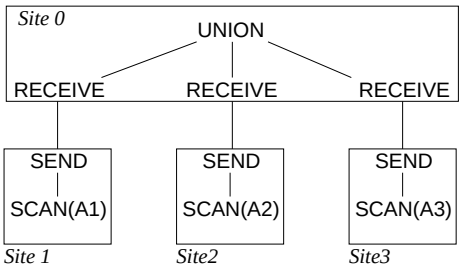


Figure 5: Example Union Plan

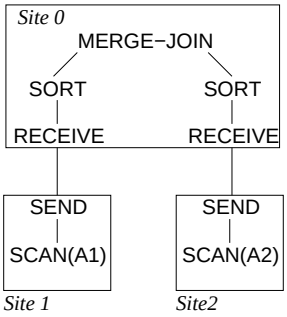


Figure 6: Example Join Plan

three *receive* operators can continuously drive the *send* and *scan* operators at Sites 1, 2, and 3 so that the *send* and *scan* 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 good 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 *receives* and *sorts* should be carried out in parallel in order to parallelize the *send* and *scan* of A and B; otherwise, the two *receive-sort* branches should be carried out one at a time in order to avoid excessive resource contention at Site 0 (i.e., disk thrashing if both sorts write concurrently to the same disk). In general, deciding which parts of a query plan should run in parallel and allocating resources such as main memory and IO bandwidth to every operator of a plan is a very difficult scheduling problem which cannot fully be solved by the query optimization approach and cost models presented in Section 2.3. The situation, for instance, gets complicated if one of the inputs of the join is produced by a more complex sub-plan at, say, Site 2. Work on scheduling and dynamic resource allocation for distributed (and parallel) databases has been described in, e.g., [CYY96, Gra96, GI97] and is beyond the scope of this survey.

2.5.5 Joins with Horizontally Partitioned Data

The logical properties of the *join* and *union* operators make it possible to process joins in the presence of horizontal partitions in a number of different ways. If, for example, Table A is horizontally partitioned such that $A = A_1 \cup A_2$, then $A \bowtie B$ can be computed in the following two ways [ESW78]:

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

If A is partitioned into more than two partitions or 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 such that all $A_i \bowtie B_j$ are empty if $i \neq j$. Consider, for example, a big company that has a `Dept` table which is partitioned by `Dept.location` in order to store all the `Dept` information at the site of the department and that has an `Emp` table which is partitioned according to the location of the `Dept` in which the `Emp` works in order to store the information of an *Emp* at the site at which the `Emp` works. A complete $\texttt{Emp} \bowtie \texttt{Dept}$ for this company can be carried out by joining the `Emp` and `Dept` partitions at every site separately. Putting it differently, the following equation holds if both the `Emp` and the `Dept` table have n partitions:

$$(\texttt{Emp}_1 \cup \ldots \cup \texttt{Emp}_n) \bowtie (\texttt{Dept}_1 \cup \ldots \cup \texttt{Dept}_n) = (\texttt{Emp}_1 \bowtie \texttt{Dept}_1) \cup (\texttt{Emp}_2 \bowtie \texttt{Dept}_2) \cup \ldots \cup (\texttt{Emp}_n \bowtie \texttt{Dept}_n)$$

2.5.6 Exploiting Replication

The $(A_1 \bowtie B) \cup (A_2 \bowtie B)$ from the previous subsection is one example of a plan in which the same data (i.e., Table B) is used in two different operations (i.e., joins). This plan is particularly attractive if Table B is replicated at the site that stores A_1 and at the site that stores A_2 because in this case, both joins can be carried out in parallel without any additional communication costs.

Another example that exploits replication in the absence of horizontal partitioning is the following Plan 1: $(A \bowtie B) \bowtie (C \bowtie B)$. This plan is equivalent to the following simpler Plan 2: $(A \bowtie B) \bowtie C$. A centralized database system would never consider a plan like Plan 1 and would always use a simple plan like Plan 2 because Plan 1 involves three (potentially expensive) join operations whereas Plan 2 involves only two. In a distributed system and, in particular, in a client-server or heterogeneous system in which the servers of A and C cannot communicate (Sections 3 and 4), however, Plan 1 might be significantly better than Plan 2 because $A \bowtie B$ and $C \bowtie B$ can be carried out in parallel and because Plan 1 might have lower communication costs than Plan 2 if copies of B are stored at the site of A and at the site of C and the results of $A \bowtie B$ and $C \bowtie B$ are fairly small [KKS98]. (In fact, the work in [KKS98] shows that a plan like Plan 1 might even be useful in centralized database systems.)

2.5.7 Semi Joins

In the SDD-1 project, semi-join programs were proposed to reduce communication costs for joins between tables (or partitions) which are stored at different sites [BGW+81]. If Table A is stored at Site 1 and Table B is stored at Site 2, then the standard way to execute such joins is to ship, say, A from Site 1 to Site 2 and then execute the join at Site 2. The idea of a semi-join 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:

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

Variants of this approach are to eliminate duplicates of $\pi(A)$ (trading additional work at Site 1 for less communication) and sending a signature file for A, a so-called bloom-hash filter, rather than $\pi(A)$ [Blo70, Bab79, VG84]. Again, the optimizer must decide which

variant to use (if any) and in which direction to carry out the semi-join 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 semi-join programs are typically not very attractive for join processing in standard (relational) distributed database systems because the additional computational overhead is usually much higher than the savings in communication costs [ML86, LC85]. Today, however, a couple of applications that involve tables with very large tuples can be found and semi-join 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, the html source of every employee's Web page, etc. In this case, it does make sense to find the target employees of a query using, say, the *age, dept_no*, etc. columns and then fetch the *picture* and other columns of the query result at the end. We will also give examples in Section 4.3.1 that demonstrate how such semi-join style techniques can be very useful to exploit the specific capabilities of sites in a heterogeneous database systems.

2.5.8 Compression

Another way to save communication costs at the expense of additional computational overhead is to make use of *compression*. If we take a look at the WWW, we can already see that compression is heavily used in various ways: JPEG for video, MPEG for pictures, PDF for text documents, or ZIP for any other large file. So, why shouldn't compression also be useful for database systems?

While there is fairly little work on compression in the context of database systems, recent work indicates that compression can indeed be very helpful to improve the performance of database systems [GS91, RHS95, GRS98, WKHM98]. In fact, most of that work concludes that compression is even attractive in centralized databases. Compression, however, is even more attractive in distributed databases because compression can save communication in addition to disk IO costs in distributed systems. Just as in the WWW, the best way to take advantage of compression in database systems is to store tables in a compressed format and then decompress them when they are needed (potentially after sending them to another site). Experiments have indicated that the best approach is to compress tables in a very fine-grained way (i.e., every attribute of every tuple of a table individually) so that only those parts of a table that are needed to execute a query need to be decompressed [GS91]. In addition, only very light-weight compression techniques such as dictionary-based compression rather than more complex techniques such as Ziv-Lempel should be used in database systems in order to be able to decompress millions of data fields very quickly [RHS95, GRS98]. Furthermore, recent experiments indicate that it is probably always too expensive to compress, ship, and then decompress data while executing a query that involves an uncompressed table because compressing a table on the fly has prohibitively high CPU costs regardless of which compression technique is used [WKHM98].

2.5.9 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 so-called pointer-based joins. Pointer-based joins occur in object-oriented and object-relational systems because foreign-keys are often implemented by

explicit references that contain the address of an object (or the address of a placeholder of an object [EGK95]) in these systems. 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 "give the `Dept` information of all `Emps` that are older than 50 years old."

Alternative ways to execute pointer-based joins have been studied in [SC90] — that paper is focussed on centralized database systems, but the basic ideas can naturally be applied to distributed (and parallel) database systems [DLM93]. The naive way to execute pointer-based joins would be to scan through, say, the `Emp` table and follow the `Dept` references of all `Emps` with `age > 50`. In a centralized database system, this naive approach has very high cost because it involves a great deal of random disk IO to fetch the individual `Dept` objects from disk. In a distributed database system, the naive approach has even higher cost because it involves a round trip message to chase the `Dept` reference of every `old Emp` in addition to random disk IO. 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., so-called *scoped* references), and this approach typically outperforms the naive approach because it avoids random disk IO 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 [SC90]. That is, we first group all `old Emps` that belong to `Depts` stored at the same site together, and then fetch the `Dept` objects for these `Emps` from that site in one batch. Like the naive approach, this approach need not recompute the matching between `Emp` and `Dept` objects; and like the value-based approach, this approach avoids random IO and excessive round trip messages (random IO can be avoided by sorting the *Dept* references). An extension, the $P(PM)^*M$ algorithm, to the algorithm(s) proposed in [SC90] was devised in [BCK98]. The $P(PM)^*M$ algorithm uses essentially the same "grouping-by-hashing" approach as proposed by [SC90], but the $P(PM)^*M$ algorithm makes sure that after the pointer-based join is complete the `Emp` tuples are in the same order as before. This is, for example, useful, if the `Emp` tuples have already the right order as needed for the query result, another join operation, or a group-by operation and it 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 [BCK98].

A special class of algorithms, so-called *object assembly*, becomes attractive if a query involves several pointer-based joins or tries to compute the transitive closure or the transitive closure up to a certain depth 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 consider systems for semi-structured data like Lore which maintain networks of similar objects with similar, yet different, structures and formats [MAG+97]. 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 as described in Section 2.3, and then join, say, all `Emps` with all `Depts` first and join the result with all `Divisions` after that in a query that involves the `Emp, Dept`, and `Division` tables. 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` could, for example, be executed as follows [KGM91, MGS+94]:

1. group `Emps` such that the corresponding `Depts` referenced by a group of `Emps` are stored on 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 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 directly be printed out as query results; `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 on the same site. That is, 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. Even better, when we visit a site in Step 2, we can expand `Emp` groups (generated in Step 1) and `Emp-Dept` groups (generated in this step) in one batch;

4. repeat Steps 2 and 3 until all `Emp` and `Emp-Dept` groups have been expanded.

As described in [MGS+94], many variants of this approach are conceivable in a distributed system; unfortunately, none of these variants have ever been implemented and no experimental performance results exist so far.

2.5.10 *Top N* and *Bottom N* Queries

Another particular kind of query are so-called *Top N* and *Bottom N* queries. 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, so-called *stop* operators can be used to isolate the top N and bottom N tuples, and query optimization issues and *stop* operator implementation issues have been discussed in [CK97, CK98]. The techniques proposed in that work were primarily developed for centralized relational database systems, but they can again be directly applied to distributed databases. To give a very simple example how these techniques would take effect in a distributed system, consider the plan shown in Figure 7. That plan computes the top ten tuples of a 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 in all no more than ten query results are produced.

Different algorithms need to be used in multi-media database systems [Fag96, CG96] or for so-called meta-searching [GCGMP97, GGM97, kI97]. As an example, consider a

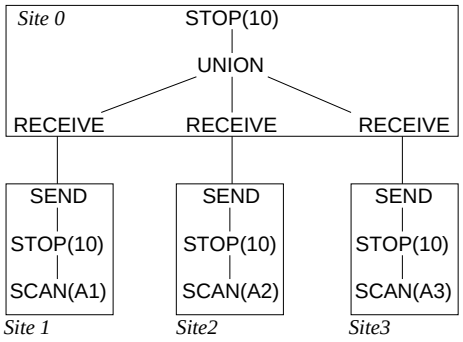


Figure 7: Example Plan for a Top N Query

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)*. Likewise, the image database returns pictures of birds in descending order of a *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 [Fag96, FW97].) 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[5], then this task can be done using the following algorithm devised in [Fag96]:

1. continuously ask the image and sound databases for the next bird with high (component) scores 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 in order to evaluate the overall scoring function for all birds returned by the two databases in the first step.

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

Similar and slightly more complicated algorithms have been proposed for meta-searching in the WWW. In this environment, people are, for instance, interested in combining the scores for Web pages returned by search engines such as AltaVista, Infoseek, Lycos, etc. in order to find Web pages with a high total score according to all search engines. The

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

algorithm from above is not applicable in this environment and different algorithms are necessary because the second (probing) step of the algorithm above cannot be carried out using today's WWW search engines [GGM97].

2.5.11 Multi-Query Optimization and Execution

Finally, we would like to briefly discuss a technique which is called multi-query optimization. The textbook query processor described in Section 2.1 takes a single query as input and tries to find the best way to execute that query given the current state of the system. An alternative is that the query processor takes a set of queries issued by, say, concurrent users or application programs and tries to find an overall good strategy to execute the whole set of queries [Sel88]. 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, and in a distributed system, these joins can be carried out at or near the sites of the queries that involve these joins.

While the benefits of multi-query optimization can be substantial, multi-query optimization has, as far as we know, not been built into any commercial product yet. The reason for this lack of acceptance is the large complexity of multi-query optimization: it is significantly more complex to isolate common subexpressions and find a good global plan for m queries than to optimize each of these m queries individually. A more light-weight variant of multi-query optimization and execution are *shared table scans*, and this technique has in fact been built into a couple of commercial products. The idea is particularly attractive in decision support systems in which there are some tables such as `Lineitem` tables which are very large and used in very many queries. In order to avoid that these large tables are scanned for every individual query, these large tables are scanned once for a group of queries and the tuples produced by this scan are then filtered, joined with other tables, and aggregated for every query individually [ZDNS98]. A similar technique is also used for video-on-demand systems and certain kinds of multi-media systems [LLG97]: again, the idea is to read multi-media objects like videos only once from disk in order to show them to a group of people and the challenge is to group people in such a way that they can be served by a single stream (considering that everybody starts watching at different times) and to buffer parts of the stream temporarily in order to deal with situations in which somebody temporarily stops his or her presentation in order to get a drink or if somebody carries out other VCR functions like `ff` or `rev`.

3 Client-Server and Multi-Tier Systems

Having described basic query processing techniques that are useful for any kind of distributed database system, we would now like to take a closer look at the currently most important class of distributed systems: systems with a client-server or multi-tier architecture. We will first describe various different client-server and multi-tier architectures that can be found in practice. Then, we will deal with one of the crucial questions of client-server query-processing: if and how to exploit the resources of client and middle-tier machines. After that, we will discuss query optimization and query execution issues and present some techniques that have become popular for client-server query processing.

3.1 Client-Server and Multi-Tier Architectures

In general, *client-server* (or *master-slave*) refers to a class of protocols that allows a site to act as a client and send a request to another site which in turn acts as a server and sends an answer as a response to this request [Tan92]. Using this general mechanism, it is possible to implement any kind of distributed database system in which every site can communicate with every other site, and every site can act as a server that stores parts of the database and as a client that executes application programs and initiates queries.

When we talk about client-server database systems, however, we will think about systems in which every site has a fixed role of always acting either as a client or as a server. One advantage of such a strict separation between client and server sites is that only server machines need to be administered (i.e., backed-up) and security issues are fairly well under control by controlling the server machines and the client-server communication links. (Note that in such a client-server environment, there are significantly more clients than servers.) 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 significantly heavier with several processors and disks and very good IO performance. Note that in such a strict client-server architecture not all sites can communicate with each other: typically, two clients do not interact and often servers do not interact either.

Many different examples for distributed database systems with such a strict client-server architecture can be found. In fact, most commercial databases (e.g., Oracle, IBM DB2, Informix, etc.) today have such a client-server architecture and, as a result, database application systems that are based on such standard commercial database systems also have a client-server architecture. Other examples include Lotus Notes, the WWW, and other Internet applications (e.g., for electronic commerce).

The strict client-server paradigm can be generalized to obtain a *multi-tier architecture*. In such an architecture, the sites are organized in a hierarchical way so that every site plays the role of a server (i.e., data source) for the sites on the level above and the role of a client (i.e., query source) for the sites on the level below: 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), and it typically cannot communicate with sites at the same or any other level. Multi-tier architectures are sometimes used to provide different functionality at different levels of the system or to improve the scalability of a system: at every level, additional sites (processes plus machines) can be added to deal with a heavier load.

As an example of a three-tier system, consider an Intranet with clients running a WWW browser and one or several WWW servers which are connected to database backend servers. In such a system, the WWW clients only communicate with the WWW servers following the client-server paradigm at that level, and the WWW servers retrieve data from the backend servers following the client-server paradigm at that level. Another prominent example of a three-tier system is SAP R/3 [BEG96]. SAP is the market-leader for business application software, and standard SAP R/3 installations consist of at least three tiers: presentation servers that drive the GUIs of the users' desktops, application servers which implement the business application logic, and database backend servers which store all the data. The Intranet example from above can be expanded to an n-tier system by considering one or more levels of proxy servers [LA94]. Another way that multi-tier systems arise is by plugging, say, two three-tier systems together; for instance,

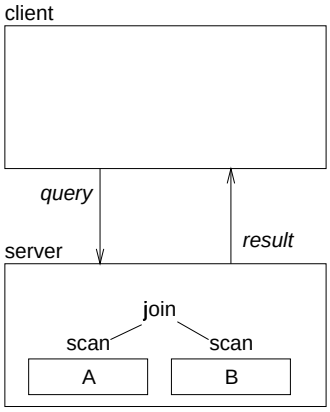


Figure 8: Query Shipping

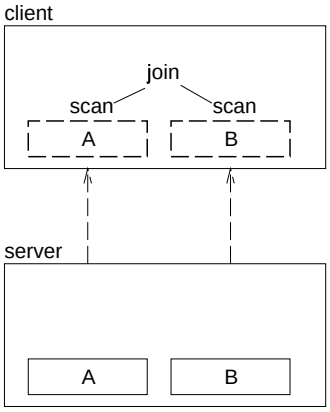


Figure 9: Data Shipping

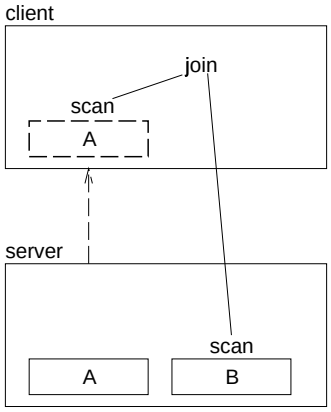


Figure 10: Hybrid Shipping

we obtain a four-tier system if WWW clients are connected to WWW servers, the WWW servers are connected to SAP R/3 application servers, and the SAP R/3 application servers are connected to database backend servers [BEG96].

3.2 Exploiting Client Resources

The essence of client-server and multi-tier systems is that the database is persistently stored by server machines and that queries are initiated by client or middle-tier machines. The question is whether to execute the queries at client and middle-tier machines or at server machines. Putting it differently, 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) [SAD+94b]. Another related question is whether and how to make use of caching; i.e., to temporarily store copies of data at client or middle-tier machines. In this section we will present and discuss the tradeoffs of alternative approaches which are all commonly used in existing systems today.

3.2.1 Query Shipping

The first approach we describe is called *query shipping*, and it is used in most relational and object-relational database systems today (e.g., IBM DB2/UDB, Oracle 8, etc.). The principle of query shipping is to execute queries as *low* as possible; i.e., to execute queries at servers and not to take advantage of caching. Figure 8 illustrates query shipping in a two-tier 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 only works if there is a middle-tier site that carries out joins between tables stored at different servers or if there are gateways between the servers so that inter-site 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 most object-oriented database systems (e.g., ObjectStore, O_2, etc.). In this approach, queries are executed at the client machine (or middle-tier machine) at which the query was initiated and data is rigorously cached at client and middle-tier machines (in main-memory or on

disk [FCL93]). 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) [DFMV90],[6] and it is possible to cache individual pages of base tables and indexes [Lom96, ZC97]. To illustrate data shipping, consider the example shown in Figure 9: in this example, 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 will use these cached copies of pages, and they will *fault in* all the pages of A and B that are not yet cached at the client.

3.2.3 Hybrid Shipping

As we will see, neither data-shipping nor query shipping is the best policy for query processing in all situations, and the advantages of both approaches can best be combined in a so-called *hybrid shipping* architecture [FJK96]. Hybrid shipping provides the flexibility to execute query operators on client and server machines, and it allows the caching of data at clients (pages of base tables and of indexes). The approach is illustrated in Figure 10; in the figure, the *scan(A)* and *join* operators are carried out at the client whereas the *scan(B)* operator is carried out at the server. Note that the *scan(A)* operator will use the client's cache as much as possible and bring to the client only those parts of A that are not in the cache already; in contrast, the *scan(B)* operator neither uses nor changes the state of the client's cache. (We will say more about the impact of query operators on caching in Section 5.) Today, hybrid shipping is used in some commercial database products (e.g., UniSQL [DJ96]), application systems like SAP R/3, database research prototypes like ORION-2 [JWKL90] and the latest version of KRISYS [DHM+98], and to some extent in heterogeneous systems like Garlic, TSIMMIS, and DISCO which we will describe in Section 4.

3.2.4 Other Hybrid Variants

In environments in which applications 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 (exploiting caching). Such an approach has, for example, been proposed as part of the KRISYS project [HMNR95], and Persistence is a product that supports this approach [KJA93]. This approach is reasonable because caching and client-side execution are particularly effective for methods that, say, repeatedly access the same objects (i.e., tuples) in order to carry out complex computations whereas queries that involve a great deal of data can often be executed most efficiently at server machines.

Another variant of hybrid shipping is used by certain decision support/OLAP 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 non-standard operations for decision support like moving averages, roll-up, drill-down, etc. [KS95, GBLP96]. 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

[6] Caching in the granularity of individual tuples has, for example, been studied in [KK94].

difference to full-fledged hybrid shipping is that not all operations can be carried out at all machines.

3.2.5 Discussion

The performance tradeoffs of query, data, and hybrid shipping have been studied in [FJK96]. Many of the effects are quite obvious. Query shipping shows good performance if the server machines are very powerful and client machines are rather slow, but it does not scale very well with the number of clients because the servers are potential bottlenecks of the system. Data shipping, on the other hand, scales well, but it can be the cause of very high communication costs for highly selective queries and if caching is not effective because a great deal of (unfiltered) base data must be shipped to the clients in such cases. Hybrid shipping has, obviously, the potential to at least match the best performance of data shipping and query shipping by exploiting caching and client resources like data shipping if that is beneficial or by behaving like query shipping otherwise. In certain situations, hybrid shipping will show better performance than data and query shipping by exploiting both client and server machines and intra-query parallelism to execute a query. The price for this improved flexibility is that query optimization is significantly more complex in a hybrid shipping system than in a query or data shipping system because the optimizer must consider significantly more options. The experiments of [FJK96] and other studies also demonstrate three (less obvious) effects for hybrid shipping systems:

- Sometimes it is better to carry out scan operators at servers and read data from the servers' disks in a hybrid-shipping system even if the data are cached at the client in order to take the best advantage of pipelined parallelism. Consider, for example, a join query that involves two tables which are stored at two different servers and assume that these tables are cached on the client's disk: the best way to execute such a query might be to read both tables from the servers' disks and to execute the join at the client rather than executing the query entirely at the client. This way, reading data from the servers' disks and join processing with the client's disk(s) do not interfere.

- Sometimes the best strategy to execute a query in a hybrid shipping system involves shipping cached base data or intermediate query results from a client to a server in order to carry out other operations (e.g., joins) of the query at that server. Such a strategy is, for example, useful in situations in which copies of some of the queried data are cached in the client's main memory and join operations can be carried out more efficiently at the server than at the client. Another reason to ship data or intermediate results from the client to the server is to take advantage of pipelined parallelism using the client and server machines.

- 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 is, for example, extensively used in SAP R/3 [BEG96, KKM98]. The advantage is that such transactions can be rolled back at clients without *hurting* the server and that the updates can be propagated to the server in one batch with fairly little overhead [OS94, BL94]. Transactions that involve updating large amounts of data (e.g., give all `Emps` a 10% salary increase), on the other hand, should of course

be carried out directly at the server(s) that store the affected data in order to avoid high communication costs to ship, say, the original `Emp` tuples from the server(s) to the client and the new `Emp` tuples back from the client to the server(s).

In all the experiments presented in [FJK96], the other hybrid variants described in the previous subsection would perform just like query shipping and show poor performance in many situations. For workloads that, say, have the right mix of C++-style methods and SQL-style queries, however, the "methods at clients and queries at servers" approach might be just as good as full-fledged hybrid shipping, and likewise, the special hybrid shipping architecture for OLAP might also be attractive for certain workloads.

3.3 Query Optimization

Having described query, data, and hybrid shipping as fundamentally different approaches for query processing in client-server systems, we will now show how query optimizers for query, data, and hybrid shipping systems can be built and describe a couple of alternative query optimization techniques which are useful for any kind of client-server and multi-tier database system and have become popular in recent years.

3.3.1 Site Selection in Client-Server Systems

From the perspective of a query optimizer, data shipping, query shipping, and hybrid shipping can be characterized by the options they allow for site selection. Every operator of a plan has a *site annotation* that indicates whether the operator is executed at a client or at a server, and Table 1 shows the possible site annotations for different classes of query operators for the three alternative approaches. 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 at which the query was initiated, but for all other operators the options vary. *Update* operators which are used at the top of plans for `update`, `insert` or `delete` statements, for example, are carried out at clients using data shipping, at servers using query shipping, and at clients or servers using hybrid shipping. For all other operators, the three approaches also vary. Using data shipping, all *scan* operators (index and table scans) are carried out at the client at which the query was initiated thereby exploiting caching; all other query operators (e.g., joins, sorts, etc.) are executed at the site at which the results of that operator are consumed so that, in effect, all query operators are executed at the client because the top-level operator, i.e., a *display* or *update* operator, is carried out at the client, too. Using query shipping, a *scan* is carried out at the server that stores the corresponding table or index (in the presence of replication, the optimizer can choose among all the sites that have replicas [see, Section 2.3.3]), and all the other operators are carried out at the or at one of the sites that produce the input of the operator so that ultimately all operators except *display* are executed by a server. Finally, hybrid shipping allows the optimizer to annotate operators in any way allowed by data or query shipping.

Note that the site annotations are *logical*: for example, a site annotation can indicate that an operator is to be carried out at the client at which the query was submitted rather than indicating that an operator is to be carried out at Machine *abc.uvw.xyz* and 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

	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, 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

Table 1: Site Selection Options for Data, Query, and Hybrid Shipping

clients so that a query need not be re-compiled for every client individually. Also note that 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, etc.) have *consumer* site annotations like data shipping.

3.3.2 Where and When to Optimize

We will now turn to 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 studied alternative approaches in an environment with many clients and one server [HF86]: they propose to carry out certain steps of query processing (e.g., parsing and query rewrite) at the client at which a query originates and other steps at the server (e.g., query optimization and plan refinement). This approach makes sense because operations like parsing and query rewrite can very well be executed at clients so that they do not disturb the server, whereas steps like query optimization require good knowledge of the state of the system (i.e., the load of the server) and should, therefore, be carried out at the server. In systems with many servers, no one server has good knowledge of the whole system at all times. In such systems, one server needs to carry out query optimization (i.e., the server located closest to the client), and this server needs to either guess the state of the network and other servers (based on statistics of the past) or try to find out 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 an extra round-trip message for every server that is potentially involved in a query.

The second question which is related to the first question is *when* to optimize a query? Again, the answer to this question impacts how accurate (in this case how recent) the information about the state of the system is that the optimizer gets. This question arises, in particular, for *canned* queries that are part of application programs and which are executed every time the application program is executed. As stated in Section 2.1, the traditional approach is to compile 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. Only when something drastic happens that makes the execution of the plan impossible (e.g., an index used in the plan is dropped or a table migrates to another site), the plan stored in the database is invalidated and a new plan is generated before the application program is executed the next

time [CAK+81]. Obviously, this approach cannot adapt to changes like shifts in the load of sites, and the compiled plans show poor performance in many situation.

A more dynamic approach was proposed by Graefe and others in [GW89, CG94] and by Ioannidis et al. in [INSS92]. The idea is to generate several plans and/or sub-plans at compile time of the application, store these alternative plans and sub-plans in the database, and choose the plan or sub-plans that best matches the current state of the system just before executing the query. An even more dynamic approach was proposed by Amsaleg et al. [AFTU96, UFA98]. Here, the idea is to start executing a (compiled or dynamically chosen) plan and then 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 in order to find a new plan for those parts of the query that still need to be carried out. Amsaleg et al. show how such a re-optimization 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 disturbed [AFTU96, UFA98]: basically, the idea is to re-order (and re-schedule) operations at the client so that the client carries out these operations while waiting for delayed data. In another paper, Kabra and DeWitt show how such a re-optimization approach helps in situations in which the initial plan shows poor performance because it was based on wrong estimates of the size of tables and intermediate query results [KD98].

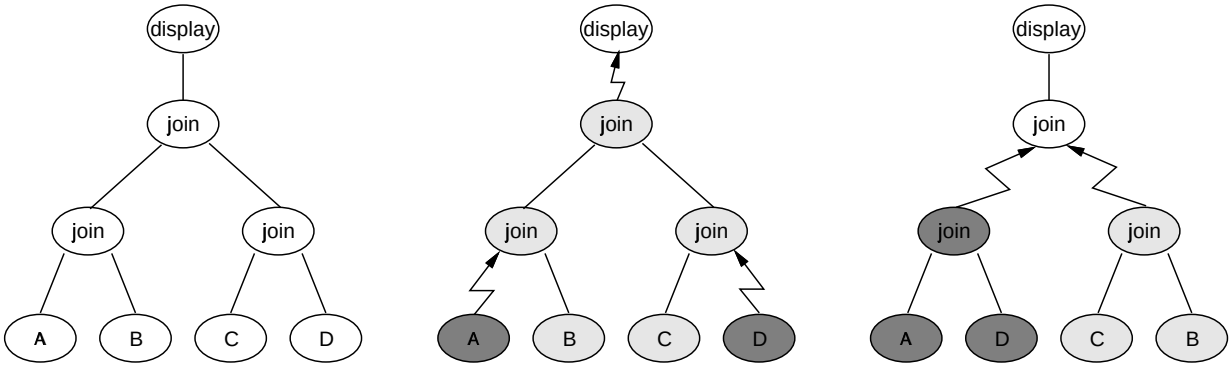
3.3.3 Two-Step Optimization

Two-step query optimization is an approach that has become popular for both distributed and parallel database systems [CL86, HS90, SAL+96, DSD95, TGHM95, HM95, GGS96]. Two-step optimization can be seen as a viable alternative for the dynamic approaches presented in the previous subsection because it carries out certain decisions just before a query is executed, and it can also be seen as an approach that reduces the overall complexity of distributed query optimization. Several variants and flavors 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 ordering and join methods, carries out access path selection, and decides whether to use pipelined query execution.

2. Every time just before the query is executed, transform the plan and carry out site selection.

Two-step optimization has reasonable complexity because both steps can be carried out with reasonable effort: the first step has essentially the same (acceptable) complexity as query optimization in a centralized database system, and the second step has reasonable complexity because it only carries out site selection. (The second step can be carried out using dynamic programming or one of the algorithms proposed in [MLR90].) Furthermore, two-step optimization is very useful to load balance a system because executing operators on heavily loaded sites can be avoided by carrying out site selection at execution time [CL86], and two-step optimization is very useful to exploit caching in a hybrid shipping system because query operators can dynamically be placed at a client



2-Step Plan at Compile-Time 2-Step Plan at Run-Time Optimal Plan

Figure 11: Increased Communication Cost Due to Two-Step Optimization

if the underlying data is cached at the client [FJK96]. On the negative side, however, such a two-step optimization approach can also produce plans with unnecessarily high communication costs. To see why, consider the example shown in Figure 11: the first plan shows the join ordering carried out in the first step of two-step optimization, the second plan shows the result of site selection in the second step, and the third plan shows an optimal plan for this query. In the second and third plan, the site annotations are indicated by the shading of the operators. Tables A and D are co-located at one server (the darkly shaded server), Tables B and C are co-located at another server (the lightly shaded server), and the result of the query must be displayed at a client workstation (the unshaded site). We can see that the second plan, obtained using two-step optimization, has higher communication costs than the third (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 costs in a distributed system.

3.4 Query Execution Techniques

Most of the query execution techniques presented in Section 2.5 make sense in a client-server environment just 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 in a hybrid shipping system) and has been implemented in almost all commercial database products. Also, it makes sense to carry out operations at the client in a multi-threaded way in a data or hybrid shipping architecture, and this is actually the way web browsers like Netscape's Navigator load individual components (e.g., text and gifs) of a web page in parallel. Furthermore, most of the other techniques described in Section 2.5 can be applied in a client-server or multi-tier environment.

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. Consider, for example, 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 query at the server that stores the `Emp` table in order to avoid costs to ship the whole `Emp` table to the client. The problem 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: one, propagate all relevant updates such as *John Doe's* new salary to the server just before starting to execute the query at the server [KGBW90]; two, 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 — such an approach can be carried out using, for example, techniques proposed in [SC92]. In either case, note that carrying out the query at the server involves additional costs, and that 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. Also note that such issues do not arise in query shipping and data shipping systems because such systems either do not support client caching and batched updates (query shipping) or carry out all query operators at the client using the latest cached versions of data (data shipping).

4 Heterogeneous Database Systems

In this section, we will describe how queries can be processed in heterogeneous database systems.[7] 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 have very different capabilities to store data, carry out database operations (e.g., joins, sorting, group-bys, etc.), and/or to 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 (pre-defined) set of queries. So, one of the challenges 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* which arises, for example, if an application is interested in the total sales and one component database uses DM as a currency while another component database uses Euro [SL90]. Furthermore, every component database has its own specific 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 concerning various aspects of the design and implementation of heterogeneous databases; in fact, there even have been excellent tutorials in the past [Ass90], and some commercial products are described in [IEE98]. 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 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 been covered, e.g., in [BGMS92].

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

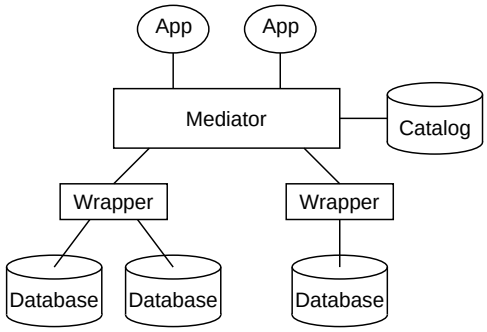


Figure 12: Wrapper Architecture of Heterogeneous Databases

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 [TRV96b], Garlic [C+95], TSIMMIS [PGMW95], Pegasus [SAD+94a], Junglee's VDB technology [GHR97], etc. (An older example is HP's MultiDatabase product [Day83].) Essentially, all of these tools have a three-tier software architecture as shown in Figure 12 so that heterogeneous systems can be seen as special instances of client-server and multi-tier systems that we studied in the previous section. Applications connect to a so-called mediator [Wie93]. The mediator parses a query, carries out query rewrite and query optimization, and incorporates a query execution engine in order to carry out, say, joins between tables stored at different component databases. The mediator also maintains a catalog in order 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), statistics for query optimization, etc. The mediator, thus, has very much the same structure as a "textbook" query processor for distributed systems which was described in Section 2. 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 which 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 hide 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 a wrapper for a sales database that uses DM as a 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 like row blocking or caching in order to improve performance. In addition, as we will see in the next subsection, wrappers also participate in query optimization.

Obviously, wrappers are fairly complex pieces of software, and it is not unusual that it takes several months to develop a wrapper. The TSIMMIS and Garlic projects have specifically addressed the question of how to make wrapper construction as cheap as possible (see, e.g., [PGGMU95, RS97]), but 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 construct a wrapper for a new component database. Also, as shown in Figure 12, it is possible that several component databases are handled by the same wrapper, and with the growing importance and demand for heterogeneous systems, it is quite likely that wrappers for many common classes of databases will be commercially available in the future. (An important prerequisite is the development of standards for the mediator-wrapper interface.)

Finally, we want to emphasize that the architecture shown in Figure 12 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 any existing wrappers. To integrate a new component database, either a new wrapper needs to be written for that component database or an existing wrapper that can deal with that component database needs to be found. As a second step, the catalog needs to be extended thereby recording the component database and its wrapper, the external schema of the component database, statistics for query optimization (if available), and adjusting the global schema of the heterogeneous system, if necessary. Also note that the architecture shown in Figure 12 is a *software* architecture: wrappers can, for example, be installed at the same or at different machines than the underlying component databases and, just as well, the mediator and wrappers may or may not be installed at the same machine. It is even possible that the mediator itself is distributed and "replicated" at different machines, and the choice of the right configuration will strongly impact the performance of the system just as in any other multi-tier database system.

4.2 Query Optimization

We will now show 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 can vary. In fact, new component databases can be added to the system at any time so that the capabilities of the component databases are not known at the time the optimizer (which is part of the mediator) is installed. As a result, the optimizer of a heterogeneous system must be generic and be able to *understand* what capabilities component databases have. In a homogeneous system, on the other hand, the query optimizer knows a-priori which operations can be carried out at which sites. (Ideally, all operations can be carried out at all sites.)

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 definition of these views in the catalog, and then see during query optimization how a query can be subsumed by the views registered in the catalog (e.g., [Qia96, LRU96]). While this approach is quite flexible, it is very difficult to implement, and this approach is not cost based so that the optimizer often does not generate good plans to execute a query. Other work proposed the use of *capability records* [LRO96] or context-free grammars to describe the capabilities of queries and the

use of a number of different (cost-based and heuristic) algorithms in order to generate plans for a query [PGH96, TRV96a]. 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 a greedy algorithm (see Section 2.3.3) in order to find a good plan for a query with reasonable effort. This approach was described in full detail in [HKWY97], and it was implemented for the first time in the Garlic system at IBM.

4.2.1 Plan Enumeration with Dynamic Programming

The idea is actually 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. That is, query optimization is carried out using the same greedy or dynamic-programming-based algorithms as described in Section 2.3.3 with the only difference that the *accessPlan* and *joinPlan* functions call planning functions defined by wrapper writers in order to enumerate subplans rather than constructing such subplans themselves.

Conceptually, planning functions can be seen as *enumeration rules*, and we will give a couple of 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 and used internally by the relational wrapper; no other wrapper nor the mediator need to know about the existence or semantics of such an *R_Scan* operator. Likewise, the relational component databases do not 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 gets as input two sets of tables (actually, the rule gets access and join plans which involve two sets of tables) stored in relational component databases and a set of join predicates for these sets of tables, and it generates a plan with an *R_Join* operator if

`plan_access`$(T, C, P) =$ **R_Scan**$(T, C, P, ds(T))$
$ds(T)$ returns the id of the relational component database that stores T.

Figure 13: Access Plan Enumeration Rule for Relational Component Databases

`plan_join`$(T_1, T_2, P) =$ **R_Join**(T_1, T_2, P)
Condition: $T_1.Source = T_2.Source$

Figure 14: Join Plan Enumeration Rule for Relational Component Databases

all tables are stored by the same relational component database. If the tables are stored at different component databases, the rule will return no plan indicating that the wrapper and its component databases cannot handle this kind of join. Again, the *R_Join* operator is used internally by the relational wrapper and plans with an *R_Join* operator will be translated into SQL queries with several tables in the `FROM` list.

To give another example, consider the BigBook database that can be accessed via 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, etc. of all matching businesses. For example, it is possible to ask for all attorneys in Arkansas. Figure 15 shows the enumeration rules defined by the wrapper for BigBook. All 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 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. Note that only `name, category, city,` and `state` predicates can be applied; to find all 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. Also note that 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 plan that returns less tuples from the BigBook database), and the other predicate would be applied in the mediator. Furthermore, note that certain queries cannot be handled although they might be syntactically correct. It is, for example, not possible to find all attorneys in the USA using the BigBook database (missing `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, note that BigBook and its wrapper are not capable of processing joins so that the wrapper provides no *plan_join* rules.

Just like wrappers, the mediator exports 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, or the mediator 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 query originate. So, an `Emp` $\bowtie$ `Dept` operation could be carried out by the mediator or by the relational component database. The optimizer

plan_access(T, C, P) = **B_Fetch**$(\{category = c, city = t\})$ *Condition:* $\{category = c, city = t\} \subseteq P$
plan_access(T, C, P) = **B_Fetch**$(\{category = c, state = s\})$ *Condition:* $\{category = c, state = s\} \subseteq P$
plan_access(T, C, P) = **B_Fetch**$(\{name = n, city = t\})$ *Condition:* $\{name = n, city = t\} \subseteq P$
plan_access(T, C, P) = **B_Fetch**$(\{name = n, state = s\})$ *Condition:* $\{name = n, state = s\} \subseteq P$

Figure 15: Plan Enumeration Rule for the BigBook Database

enumerates both alternatives by calling the mediator and wrapper join enumeration rules and the overall cheaper plan wins.

The full details and a description of a more elaborate example can be found in [HKWY97]. Having presented the basic idea, we will just briefly summarize the major advantages of this approach. One, this approach relies on well-established distributed database technology; the use of dynamic programming (or greedy algorithms) 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 turn their products into products for heterogeneous database systems. Two, this approach is very flexible, and it is possible to model the capabilities of the component database very accurately; although we did not show this here, it is, for example, possible to take gateways between different component databases or replication of tables at different component databases into account. Three, it is quite easy to implement the planning functions of a new wrapper and, thus, to integrate new wrappers and component databases. 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. Four, it is possible to define only very simple enumeration rules for a wrapper in order to integrate a new wrapper with new component databases at the beginning and then extend the wrapper and add more sophisticated enumeration rules once the wrapper is operational in order to generate plans that fully exploit the capabilities of the component and get good performance. In fact, some very simple generic rules exist that can be used to integrate any new wrapper and component database [HKWY97]. Five, 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 Costing

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 2.3.2 can be applied 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, etc. The challenge is to estimate the cost (or response time) of wrapper plans which 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. Principally, there are three alternative approaches that differ in the accuracy of the estimates and in the amount of effort that needs to be invested by wrapper writers. We will briefly describe these three approaches in the following. Unfortunately, there has, to the best of our knowledge, been no work in the literature that compares these three approaches for important classes of wrappers and component databases.

Calibration Approach The first approach is called the *calibration approach*. The idea is to define a *generic* cost model for all wrappers, and then adjust certain parameters of this cost model for every individual wrapper and component database by executing a set of sample queries. This way, 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, for example, small for very fast component databases and quite large for slow component databases or component databases that are only reachable using 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., [DKS92, ZL94, GGT96]). The generic cost models described in that work are significantly more complex than the simple example we gave above; for example, these cost models typically define special cost formulae for single table queries, multi-table queries, indexed and non-indexed queries, etc. The big advantage of the calibration approach is that wrapper writers need not worry much about costing issues when they construct a new wrapper and/or integrate a new component database into the heterogeneous database: the generic cost model is pre-defined as part of the mediator and the calibration of the generic cost model for a new wrapper and component database can be carried out (semi-) automatically using the sample 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 [DKS92, ZL94, GGT96], 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, wrapper writers do not only provide enumeration rules as described in the previous subsection, but they also provide a set of cost formulae: one cost formula is associated to every enumeration rule in order to estimate the cost of the wrapper plan(s) generated by this rule. Obviously, the big advantage of this approach is that the cost of all wrapper plans can be modeled as accurately as desired. On the negative side, however, this "do it yourself" approach puts a heavy burden on wrapper writers. To combine the advantages of the calibration approach and this "do it yourself" approach, Naacke et al. [NGT98] proposed an approach in which costing is done by default using the calibration approach (i.e., generic cost model plus query sampling) and wrapper writers 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.

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 [ACPS96]. In this approach, the system would, for example, observe that the last three wrapper plans that involved Tables A and B had costs of, say, 10 secs, 20 secs, and 9 secs, and the system would then estimate that the next wrapper plan involving A and B would cost 13 secs. (Similar and more sophisticated ideas of query feedback have also been studied in the standard relational context [CR94a].) Like the calibration approach, this approach releases wrapper writers from the burden of thinking about costing issues, but this approach can obviously be very inaccurate resulting in poor plans and, thus, in poor performance. One particular advantage of this approach is that it automatically adapts to changes that impact the cost of operations in the component database (e.g., growing tables, hardware upgrades, different load situations). Also, note that this approach can be combined with the other two approaches by estimating the cost of plans for certain wrappers using a wrapper specific cost model and by using this or the calibration approach for other wrappers.

4.3 Query Execution Techniques

We will now discuss two techniques which have become popular to execute 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 2.5 and 3.4 are applicable and useful in the mediator of a heterogeneous system (e.g., batching updates or multi-threaded query execution). Keep in mind, however, that the wrappers and component databases have limited capabilities and significantly restrict the possible ways to execute a query. For instance, we cannot expect that every component database is capable of participating in a semi-join program with duplicate elimination, and two component databases typically cannot interact with each other (unless there is a gateway). Also, keep in mind that it is (usually) not possible to place query operators at component databases; instead, operators must be translated into queries which are understood by the component databases' API.

4.3.1 Bindings

The first technique we describe simulates an (indexed) nested-loop join in a heterogeneous system. (In System R^*, a similar technique was called *fetch as needed* [ML86].) 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 database 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 one by one (or in blocks using row blocking) to the mediator. 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 value of the x field of the current tuple of A.

This approach corresponds to an indexed nested loop join in classic database environments if component database D_2 has an index on $B.y$ (which it would usually have if it allows bindings for $B.y$ as part of its interface). 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, and B is very large. Even if this approach is not very efficient (e.g., A is large), this approach might be necessary because it might be the only possible way to execute $A \bowtie B$; for example, BigBook only allows queries that restrict the business category and city of companies (using predicates with bindings) so that it would not be possible to, say, ship the whole BigBook `address` information to the mediator and carry out joins with other tables in the mediator.

Certain component databases accept blocks of tuples as parameters; e.g., component databases which are managed by a standard relational database product. 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 round-trip messages. In our 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 and "?" would be instantiated with a block of tuples from A or the whole A table. Because it reduces the number of round-trip messages, this approach is usually significantly faster than the tuple-at-a-time approach and should, therefore, always be used, if it is applicable. This approach corresponds roughly to a block-wise nested-loop join or to a special kind of semi-join program (depending on whether all columns of A are passed down to D_2 or only the x column is passed to D_2).

4.3.2 Cursor Caching

Exploiting bindings, the mediator submits the same query (with different parameters) many times. To implement the binding-based nested loop join, for example, the same query is submitted for every tuple of A and, in fact, only four different kinds of queries can be submitted to the BigBook database over the web in any case. The idea of *Cursor Caching* which can be implemented as part of a wrapper is to keep the context of a binding query (i.e., the connection to the component database and a plan for the binding query) in order to reduce the overhead of submitting the same query to the same component database with different parameters repeatedly. For component database systems that understand JDBC, for example, cursor caching can be implemented by using JDBC's `prepareStatement` command to, say, optimize once and for all times the query and then the `set` command to pass the binding parameter(s) and the `executeQuery` command to execute every instance of the query. Cursor caching is another technique which is extensively used by database application systems like SAP R/3 [DHKK97], and similar ideas have also been integrated into several DBMS products (e.g., Oracle8 [LJJC98]).

The danger of cursor caching is that the best query plan might depend on the value of the binding parameter so that it is not advantageous to use the same plan for every instance of a query [DHKK97]. As an example, consider the following query to a relational component database with a (secondary) index on `Emp.age`:

```
SELECT *
FROM Emp e
WHERE e.age > ?;
```

If we ask for all `Emps` that are older than 80 years, the best plan to answer this query probably involves using the `Emp.age` index. If we ask for all `Emps` that are older than 20 years, on the other hand, using the `Emp.age` index might result in terrible performance because of random disk IO.

4.4 Outlook

While query processing for homogeneous and client-server query processing as described in Sections 2 and 3 is fairly well understood, this is not (yet) totally true for heterogeneous systems. As we saw in this section, writing wrappers is often still a tedious task and query optimization is more difficult because the component databases are autonomous, have different capabilities, and induce costs which are hard to predict. Nevertheless, we are going to see more and more heterogeneous database systems in practice. The WWW with its millions of small databases is one area that sparks the development of (global) heterogeneous systems, and as we saw in the introduction, some of today's monolithic systems are going to be broken down into individual components and end up as heterogeneous databases in order to market individual components of the system individually and, maybe, integrate other components by third-party vendors. (SAP and Microsoft are companies that have this strategy [KKM98].) The development of interoperability standards such as CORBA and OLE (and OLE DB) will make data processing in distributed, heterogeneous environments become even more important, and commercial products from database vendors (e.g., IBM's Garlic [C+95] or HP's Pegasus [SAD+94a]) as well as new

start-up companies (e.g., Junglee [GHR97]) are appearing on the market-place. Furthermore, (industrial and academic) research projects are developing more and more ways in which database and application components interoperate (e.g., [BRS96, RSS98]).

What we discussed in this section is only a small fraction of all the work in this area, and there is definitely a great deal of work to come. However, we believe that we showed the most important trends. When designing a heterogeneous database, the goal is to hide as much heterogeneity as possible and use as much existing distributed database technology as possible: wrappers hide the details of the component database's schema, data model, and query interfaces; mediators process queries fairly much like any other distributed (client-server/hybrid shipping) database system; and an extended query optimization approach makes it possible to carry out query optimization (potentially) just as well as in any other distributed system. Only a few simple query execution techniques such as *bindings* and *cursor caching* are required in addition to the query execution techniques described in Sections 2.5 and 3.4 in order to take the best advantage of the capabilities of component databases and reduce the overhead to interact with component databases.

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 angle 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 which kind of queries might be carried out at which 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., [Ape88]). 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 keep up; (3) the complexity of all reasonable models for static data placement is too big for most of today's distributed systems because these systems involve hundreds of sites and process dozens of different kinds of queries with many different types of data (tables, indexes, multi-media objects, etc.). In this section, we will, therefore, focus on *dynamic data placement* approaches that 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 overall data placement with every move.

As in the rest of this paper, we will not deal with concurrency control or consistency issues and fully concentrate on methods that decide where to store copies of data. Concurrency control issues which are relevant for dynamic data placement and the techniques we describe in this section have, for example, been addressed in [DGMS85, Fra96, Lom96, ZC97]. Also we will concentrate on deciding where copies of (parts of) base tables and indexes should be placed, and we will ignore the issue of cleverly placing copies of catalog

	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

Figure 16: Differences Between Replication and Caching

information at different sites in order to quickly find these copies at all times; such issues have, for example, been specifically addressed in [EKK97] (see Section 2.4).

5.1 Replication vs. Caching

First, we would like to establish some terminology and give a brief overview. 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 which are kept at servers with the purpose to better meet 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), 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 places where the data is needed and caching exploits client and middle-tier machine resources which might remain unused without caching.

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 at a client or middle-tier 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 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 which 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 [Fra96, FCL97]. 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 (more interesting) data using a replacement policy such as LRU (if the copies have not already been removed from the cache because of updates).

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 hereof and moves it to the target server. Caching, on the other hand, is a by-product of query execution: when, for example, a table scan (or index scan) is executed at a client, the client will *fault in* all the pages of the table (or index) that the client has not cached, and after the scan is complete, the client will keep all the used pages of the table (or index) in its cache, if the cache is large enough (Section 3.2.2). Putting it differently, replication can take effect at servers even if no queries are actually carried out at these servers, whereas the cache of a client will be empty if no queries are executed at that client. As a consequence, we will show in this section that caching decisions need to be made by the query processor while replication decisions can be made by a separate component which 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, while caching makes it possible to cheaply access data when it is used repeatedly by a specific client or middle-tier machine. In fact, we can see both replication (or mirroring) and caching happening in the WWW and Internet. Another comment we would like to make 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 completely concentrate on the performance implications of replication and caching. Finally, we want to say that we see *migration* as 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. This perspective will become clearer when we look closer at a (dynamic) replication algorithm in the next subsection.

[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 [FCL93].

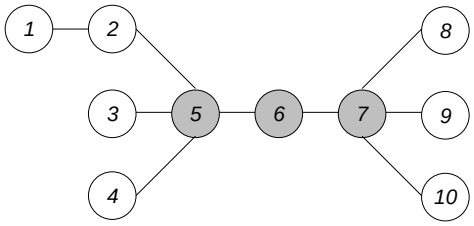


Figure 17: Replication Scheme of the ADR Algorithm

5.2 Dynamic Replication Algorithms

Several dynamic replication algorithms have been proposed in the literature [GS94, BC96, FNY93, SAB+96, CABK88, WJH97]. These algorithms can roughly be classified into two groups: (1) algorithms that try to reduce *communication costs* in a WAN by moving copies of data to servers which 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 of servers in a LAN or in an environment in which communication is fairly cheap. 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, which is targeted to reduce communication costs and works particularly well in tree-shaped networks. The ADR algorithm is attractive because it is very simple, has provably good performance in certain environments, and can easily be integrated into most distributed systems. Also, the ADR algorithm is a nice representative of a common class of dynamic replication algorithms which are based on similar principles (e.g., [GS94, BC96]). We will discuss a different class of dynamic replication algorithms which are based on completely different ideas 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 hereof) should be a connected subgraph in order to minimize the communication costs in a tree-shaped (hierarchical) network.

Consider, as an example, the network shown in Figure 17 which has ten servers and in which an object is replicated at Servers 5, 6, and 7 (depicted by the shading in Figure 17). The observation is that even if the object is very rarely accessed by clients of Server 6, the object should nevertheless be replicated at Server 6, if the object is replicated at Servers 5 and 7. When the object is updated by a client of, say, 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 costs in this case. Likewise, Server 6's copy of the object can be kept consistent with no additional communication costs if the update originates at a client of Server 7, 8, 9, or 10, and under no circumstance would Server 6's copy of the object hurt if the object is read anywhere in the system.

Based on this observation, 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 expand (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 which 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 were propagated to that copy than the copy was 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, the contraction test must not be carried out if the replication scheme consists of only one server in order to avoid that the only copy of the object is dropped.

5.3 Cache Investment

We will now turn to caching and a method called cache investment [FK97]. Like the ADR algorithm, cache investment keeps statistics and only establishes copies of data at clients if these copies promise to be beneficial. Since replication and caching are different, however, there are a number of subtle albeit 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*) to load a client's cache with parts of a table and/or index, and (2) compute the *benefits* in cost 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 (if the data is not cached there already) and subsequent queries can be executed using the cache. Queries that involve data that should not be cached should preferably be executed at servers without invoking extra costs for faulting in data and without replacing useful data by useless data in the client's cache.

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;
```

Recall from Section 3.2.3 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, executing this query 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, resulting in overall communication costs of 30 pages. If the query is executed at the server, only a couple of (thin) `Emp` result tuples on which the `name` and `manager` fields have been projected out need to be shipped from the server to the client. Using row blocking as described in Section 2.5.2, let us assume that 10 pages need to be shipped from the server to the client in this plan so that the optimizer would (normally) decide to execute the query at the server.

In this example, cache investment kicks in if the client repeatedly asks queries which involve `Emps` with high salary. In this case, cache investment advises the optimizer at one point to generate a plan that executes the index and table 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 salary can be carried out at the client with no communication costs. Without cache investment, the optimizer would execute all queries at the server, no data that could be used to answer subsequent queries would be cached at the client, and every query like the example query above would have communication costs of 10 pages. 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, e.g., 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 finds out that after three "high salary" queries the *benefits* of caching outweigh the *investment* and will, thus, advise the optimizer to generate a suboptimal plan to load the cache with the `Emp` data.

There are quite a few more details that need to be taken into account to make cache investment work properly; e.g., the exact interaction of cache investment and query optimization, dealing with updates, limitations in the size of a client's cache, light-weight strategies to estimate the *benefits* and *investment* of caching, cost formulae for clustered and unclustered indexes, 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 [FK97] so that we will not discuss them here. To conclude, we just want to briefly highlight again the differences between caching with cache investment and replication with the ADR algorithm:

- Caching is fine-grained making it possible to cache only few, frequently used pages of large tables or indexes as in the example above. Shipping, caching, and keeping consistent copies of the whole `Emp` table at all clients that frequently ask for `Emp` information is just not practical in environments with hundreds or thousands of clients.

- The investment to establish a copy is significantly lower with caching than with replication because caching takes effect when data must be processed 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.

- Recall from Section 5.1, however, that probably both replication with the ADR algorithm (or some other algorithm) and caching with cache investment (or some similar technique) should be employed as 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 *kind* 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 indexes or parts hereof. 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; see, for example, [RCK+86, KB94, DFJ+96, DRSN98, DHM+98]. The most prominent example of commercial systems that materialize and/or cache views are data warehouses [Wid95]. Data warehouses are typically established for decision support in companies or as product catalogs and classified ads for, e.g., electronic commerce on the web. They are usually constructed in a three-tier environment: the data warehouse is located in the middle-tier, it is connected to one or several data sources, and it keeps materialized views over the base data stored at the data sources in order to answer queries from clients without interacting with the data sources. In fact, already a huge industry has formed around this concept, and data warehousing definitely deserves more attention than we give it in this small section. From our (narrow) perspective, however, we see a data warehouse, the data sources, and the clients as part of a distributed system in which views are materialized or cached in the warehouse.

Compared to 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 might, for example, 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, is significantly more complex to implement. One, keeping cached or materialized views consistent in the presence of updates is complex and often expensive [Rou91, QW97], 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. Two, the ADR algorithm can obviously not be applied to decide

which views to materialize and algorithms that carry out such decisions are just beginning to emerge [HRU96, YKL97, SSV96]. Cache investment can principally be applied, but there is an explosion in the number of "what-if" analyses that need to be carried out for every query which makes a naive application of cache investment impractical.

Query optimization is also more complicated and more expensive in the presence of cached and/or materialized views. First of all, the optimizer must find out whether a cached or materialized view *subsumes* the query result (or an intermediate query result). That is, the optimizer must decide whether the cached or materialized view contains all the (intermediate) result tuples of the query; only in this case, the cached or materialized view can be used to evaluate the query. Solving the subsumption problem involves looking at the predicates, tables, and aggregate functions of the view and query definition, and a number of alternative algorithms to solve this problem have been proposed in the literature (e.g., [LMSS95]). Also, it was shown that the subsumption problem is, in general, undecidable given the complex semantics of SQL, so that the optimizer must be conservative and decide that a cached or materialized view is inapplicable, if in doubt. Once, the optimizer has found the set of applicable cached and materialized views for a query, the optimizer must decide which views to actually use because obviously some views might be more attractive than others, and it is possible that the query should be evaluated using the base data even if cached and materialized views are applicable [CR94b]. Carrying out this decision, however, is fairly simple because the optimizer can enumerate a *read(view)* plan for all applicable views just like other *access plans* and *join plans* and carry out cost-based optimization using dynamic programming or a greedy algorithm as described in Section 2.3.3. 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. Putting it differently, 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 4.2.1.

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 and other network and communication technology have sparked a large number of new applications and have led to systems with an ever growing number of clients and servers for which the state-of-the-art query processing approach presented in the previous sections might be too rigid. In this section, we will describe recent trends and developments that try to overcome the limitations of state-of-the-art query processing approaches. Specifically, we will give a brief overview of economic models for distributed query processing and dissemination-based information systems, and these developments indicate that completely new scenarios for distributed data processing might become attractive in the near future. Neither economic models nor data dissemination are mature technologies yet, but they definitely represent

approaches to watch out for in the next couple of years.

6.1 Economic Models for Distributed Query Processing

At several points, we already described or gave pointers to techniques that are based on economic principles. Cache investment, for example, is based on the simple principle that copies of data are moved to a client only if the additional cost (investment) of moving the copies to the client is lower than the expected return of investment (benefit) of caching the data at the client (Section 5.3). Similarly, the resource allocation work referenced in Section 2.5 is based on economic models (i.e., [FNS91, YC91, DG95]). In fact, a large variety of economic models for various aspects of distributed computing (e.g., resource allocation in general, load balancing, flow control, quality of service, etc.) has been studied since the mid eighties, mostly in the context of distributed operating systems [FYN88, WHH+92, FNY93], and a good survey is given in [FNSY96]. In general, the motivation to use an economic model is that distributed systems are too complex to be controlled by a single centralized component with some kind of 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, and the hope is that the specific needs of all the individual clients are best met if all servers act this way.

Concentrating again on distributed databases, Mariposa is the first system in which all sites make all their decisions based on economic principles, and Mariposa is the first distributed database system that carries out an *auction* to process queries [SAL+96]. In a nutshell, query processing in Mariposa works as follows (more details can be found in [SAL+96]):

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 a minute.

2. Every query is then handled 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, e.g., 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, say, execute one or several of the join operations specified in the broker's plan is free to give bids of the form:

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

   That is, a server indicates with such a bid that it will be willing to execute Operator o for p dollars in t seconds, and this offer is valid until the expiration date x.

4. The broker collects all bids and makes contracts with servers in order to execute the queries. Doing so, the brokers 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,

say, only \$1.00 to servers, the borker will pursue this way and keep \$4.00 of the budget as profit. If the query cannot be evaluated with reasonable cost within a 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 rejects the query. In this case, the client will have to raise the budget or loosen the response time goals or just be happy without the answer.

At first glance, Mariposa's query processing approach does not appear to be much different from the techniques we presented in Sections 2 and 3: essentially, Mariposa carries out two-step optimization as described in Section 3.3.3 and momentarily heavily loaded servers that are avoided in Mariposa because they make no or only expensive bids will be avoided in an ordinary two-step or dynamic optimization approach as well because the cost factors associated to such servers will be very high. The beauty of Mariposa is that different servers can flexibly establish different bidding strategies in order to achieve high revenue. It is, for instance, possible that a server specializes on, say, 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, and 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 profit by buying and selling copies of data [SAB+96]. The football WWW server located in Paris, for example, cannot possibly handle all requests from all over the world during the World Cup finals. That server will allow other servers, say, in Brazil or Nigeria to replicate the results of the football matches and get additional revenue for selling the original copy of the `Results` table and for propagating all updates. Servers in Brazil and Nigeria will try to buy copies of the `Results` table so that they can bid to execute 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 Mariposa is going to be commercially available soon, we would like to note once more that 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 in order 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, we assumed a data delivery model in which clients request data stored at servers. In this model, the clients are the active part and initiate query processing activities. Lately, there has been a big hype about *push technology*. In this model, servers are the active part and disseminate data to clients before the clients request the data. An early incarnation of *push* is TeleText provided by, e.g., most European TV stations since the mid eighties. Furthermore, both Netscape's Navigator and Microsoft's Internet Explorer provide features to allow users to (passively) listen to data which is disseminated by WWW servers. Pointcast's product which displays news (and commercials) on PC

screens based on a user's profile of interests has been one of the most spectacular product releases in this domain. A nice overview of these and other push-based systems and the whole hype about push is given in [FZ98].

One reason for this hype is the psychological aspect that most people would like to get all the information they are interested in with virtually no effort. In addition, however, there are a number of technical aspects which are worth considering; in particular, if data push is carried out in networks that support broadcasts or *1:N* multicasts. First of all, broad and multicasting is a way to solve some of the scalability problems of the classic, pull-based data delivery approach by batching the requests for the same object of multiple clients and processing them in one wash, rather than processing every client request individually and doing the same thing several times for different clients, as is done by WWW servers and most other information systems today [AF98]. In conjunction with push, broad and multicasting makes it possible that clients do not even need to explicitly request *hot* data which is automatically disseminated, thereby avoiding costs to send and process request messages [AFZ97]. Other interesting aspects are client-side caching in push-based and data broadcasting systems [AAFZ95, AFZ96], and multi-tier architectures for data dissemination [FZ98].

Unfortunately, SQL-style query processing has not yet been studied very well in the context of push-based systems or systems that employ broad and multicasting. One way to exploit multicasting capabilities of the network would be to make use of multi-query optimization and execution as described in Section 2.5.11 and some of the other techniques described in Section 2.5 would also benefit from such multicasting capabilities (e.g., joins with horizontally partitioned data). Also, there are some environments in which client tools periodically ask the same queries in order to inform users of, say, the latest football results or stock prices; from the perspective of the user, this appears as "push" (from the tool), but it is in fact a "pull" by the tool, and it can be supported using the techniques presented in Section 3. In all, we must admit, however, that push and multicasting technology have had very little impact on the development of SQL-style query processing so far, and pure push technology and SQL-style query processing as described in this paper do not seem to mix well.

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. Also, 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 multi-tier architecture, and many complex systems are composed of several sub-systems from potentially different vendors with heterogeneous data processing capabilities and APIs.

This paper gave an overview of the state of the art in distributed query processing. We presented a series of techniques covering all aspects of query processing: from basic techniques for query optimization and query execution to more specialized techniques for certain classes of client-server and heterogeneous database systems, and we showed how

caching and replication techniques can be used in large systems with many clients and servers. Combined, this set of techniques should be sufficient to support most of today's database applications. We also discussed recent trends that might dramatically change the way some distributed systems will be built in the future.

While we do believe that most issues of distributed query processing are well understood, we do want to make a couple of final remarks. One, no vendor has implemented all or jus a significant portion of the techniques described in this paper. Conceptually, the pieces fit together fairly well, but it is nevertheless not always easy to integrate a new technique into an existing system. It is, for example, 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. Two, 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 system can directly benefit from them, but in fact, several of the techniques presented in this paper have been implemented as part of the SAP R/3 business application system [KKM98] 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, and the situation gets worse considering the current trend to design and market application and database management modules that can freely be plugged together and may interact in unpredictable ways.

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

References

[AAFZ95] S. Acharya, R. Alonso, M. Franklin, and S. Zdonik. Broadcast disks: Data management for asymmetric communication environments. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 199–210, San Jose, CA, USA, May 1995.

[ACPS96] S. Adali, K. Candan, Y. Papakonstantinou, and V. S. Subrahmanian. Query caching and optimization in distributed mediator systems. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 137–148, Montreal, Canada, June 1996.

[AF98] D. Aksoy and M. Franklin. Scheduling for large-scale on-demand data broadcasting. In *Proc. IEEE INFOCOM Conf.*, San Francisco, CA, USA, March 1998.

[AFTU96] L. Amsaleg, M. Franklin, A. Tomasic, and T. Urhan. Scrambling query plans to cope with unexpected delays. In *Proc. of the Intl. IEEE Conf. on Parallel and Distributed Information Systems*, Miami Beach, FL, USA, December 1996.

[AFZ96] S. Acharya, M. Franklin, and S. Zdonik. Prefetching from a broadcast disk. In *Proc. IEEE Conf. on Data Engineering*, New Orleans, LA, USA, 1996.

[AFZ97] S. Acharya, M. Franklin, and S. Zdonik. Balancing push and pull for data broadcast. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 183–194, Tucson, AZ, USA, May 1997.

[Ape88] P. Apers. Data allocation in distributed DBMS. *ACM Trans. on Database Systems*, 13(3):263–304, September 1988.

[Ass90] Special issue on heterogeneous databases. ACM Computing Surveys, Vol 22, No 13, September 1990.

[ASU87] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1987.

[Bab79] E. Babb. Implementing a relational database by means of specialized hardware. *ACM Trans. on Database Systems*, 4(1):1–29, March 1979.

[BC96] A. Bestavros and C. Cunha. Server-initiated document dissemination for the WWW. *IEEE Data Engeneering Bulletin*, 19(31):3–11, September 1996.

[BCK98] R. Braumandl, J. Claussen, and A. Kemper. Evaluating functional joins along nested reference sets in object-relational and object-oriented databases. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, New York, USA, August 1998.

[BEG96] R. Buck-Emden and J. Galimow. *SAP R/3 System, A Client/Server Technology.* Addison-Wesley, Reading, MA, USA, 1996.

[BGMS92] Y. Breitbart, H. Garcia-Molina, and A. Silberschatz. Overview of multidatabase transaction management. *The VLDB Journal*, 1(2):181–293, 1992.

[BGW+81] P. Bernstein, N. Goodman, E. Wong, C. Reeve, and J. Rothnie. Query processing in a system for distributed databases (SDD-1). *ACM Trans. on Database Systems*, 6(4), December 1981.

[BL94] P. Bogle and B. Liskov. Reducing cross domain call overhead using batched futures. In *Proc. of the ACM Conf. on Object-Oriented Programming Systems and Languages (OOPSLA)*, pages 341–354, Portland, OR, USA, October 1994.

[Blo70] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13:422–426, 1970.

[BRS96] S. Blott, L. Relly, and H.-J. Schek. An open storage system for abstract objects. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 330–340, Montreal, Canada, June 1996.

[C+95] M. Carey et al. Towards heterogeneous multimedia information systems. In *Proc. of the Intl. Workshop on Research Issues in Data Engineering*, March 1995.

[CABK88] G. Copeland, W. Alexander, E. Boughter, and T. Keller. Data placement in bubba. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 99–108, Chicago, IL, USA, May 1988.

[CAK+81] D. Chamberlin, M. Astrahan, W. King, R. Lorie, J. Mehl, T. Price, M. Schkolnik, P. Selinger, D. Slutz, B. Wade, and R. Yost. Support for repetitve transactions and ad hoc queries in System R. *ACM Trans. on Database Systems*, 6(1):70–94, March 1981.

[CBB+97] R. Cattell, D. Barry, D. Bartels, M. Berler, J. Eastman, S. Gamerman, D Jordan, A. Springer, H. Strickland, and D. Wade. *The Object Database Standard: ODMG 2.0.* The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann Publishers, San Mateo, CA, USA, 1997.

[CG94] R. Cole and G. Graefe. Optimization of dynamic query evaluation plans. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 150–160, Minneapolis, MI, USA, May 1994.

[CG96] S. Chaudhuri and L. Gravano. Optimizing queries over mulitmedia repositories. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 91–102, Montreal, Canada, June 1996.

[CK97] M. Carey and D. Kossmann. On saying "enough already!" in SQL. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 219–230, Tucson, AZ, USA, May 1997.

[CK98] M. Carey and D. Kossmann. Reducing the braking distance of an SQL query engine. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, New York, USA, August 1998.

[CL86] M. Carey and H. Lu. Load balancing in a locally distributed database system. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 108–119, Washington, USA, 1986.

[Com79] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[CP84] S. Ceri and G. Pelagatti. *Distributed Databases – Principles and Systems.* McGraw-Hill, Inc., New York, San Francisco, Washington, D.C., 1984.

[CR94a] C. Chen and N. Roussopoulos. Adaptive selectivity estimation using query feedback. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 161–172, Minneapolis, MI, USA, May 1994.

[CR94b] C. Chen and N. Roussopoulos. The implementation and performance evaluation of the ADMS query optimizer: Integrating query result caching and matching. In *Proc. of the Intl. Conf. on Extending Database Technology (EDBT)*, volume 779 of *Lecture Notes in Computer Science (LNCS)*, Cambridge, United Kingdom, March 1994. Springer-Verlag.

[CS94] S. Chaudhuri and K. Shim. Including group-by in query optimization. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 354–366, Santiago, Chile, September 1994.

[CYY96] M.-S. Chen, P. Yu, and T.-H. Yang. Optimization of parallel execution for multi-join queries. *IEEE Trans. Knowledge and Data Engineering*, 8(6), December 1996.

[Dat95] C. J. Date. *An Introduction to Database Systems.* Addison-Wesley, Reading, MA, USA, sixth edition, 1995.

[Day83] U. Dayal. Processing queries over generalization hierarchies in a multidatabase system. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, Florence, Italy, 1983.

[DFJ+96] S. Dar, M. Franklin, B. Jónsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 330–341, Bombay, India, September 1996.

[DFMV90] D. DeWitt, P. Futtersack, D. Maier, and F. Velez. A study of three alternative workstation server architectures for object-oriented database systems. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 107–121, Brisbane, Australia, August 1990.

[DG92] D. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *Communications of the ACM*, 35(6):85–98, June 1992.

[DG95] D. Davison and G. Graefe. Dynamic resource brokering for multi-user query execution. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 281–292, San Jose, CA, USA, May 1995.

[DGMS85] S. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in partitioned networks. *ACM Computing Surveys*, 17(2):341–370, September 1985.

[DHKK97] J. Doppelhammer, T. Höppler, A. Kemper, and D. Kossmann. Database performance in the real world: TPC-D and SAP R/3. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 123–134, Tucson, AZ, USA, May 1997.

[DHM+98] S. Deßloch, T. Härder, N. Mattos, B. Mitschang, and J. Thomas. KRISYS: Modeling concepts, implementation techniques, and client/server issues. *The VLDB Journal*, 7(2):79–95, April 1998.

[DJ96] A. D'Andrea and P. Janus. UniSQL's next-generation object-relational database management system. *ACM SIGMOD Record*, 25(3):70–76, September 1996.

[DKS92] W. Du, R. Krishnamurthy, and M.-C. Shan. Query optimization in heterogeneous DBMS. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 277–291, Vancouver, Canada, August 1992.

[DLM93] D. DeWitt, D. Lieuwen, and M. Mehta. Parallel pointer-based join techniques for object-oriented databases. In *Proc. of the Intl. IEEE Conf. on Parallel and Distributed Information Systems*, San Diego, CA, USA, January 1993.

[DRSN98] P. Deshpande, K. Ramasamy, A. Shukla, and J. Naughton. Caching multidimensional queries using chunks. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 259–270, Seattle, WA, USA, June 1998.

[DSD95] W. Du, M.-C. Shan, and U. Dayal. Reducing multidatabase query response time by tree balancing. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 293–303, San Jose, CA, USA, May 1995.

[EGK95] A. Eickler, C. Gerlhof, and D. Kossmann. A performance evaluation of OID mapping techniques. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 18–29, Zürich, Switzerland, September 1995.

[EKK97] A. Eickler, A. Kemper, and D. Kossmann. Finding data in the neighborhood. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 336–345, Athens, Greece, August 1997.

[ESW78] R. Epstein, M. Stonebraker, and E. Wong. Query processing in a distributed relational database system. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 169–180, 1978.

[Fag96] R. Fagin. Combining fuzzy information from multiple systems. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, pages 216–226, Montreal, Canada, 1996.

[FCL93] M. Franklin, M. Carey, and M. Livny. Local disk caching for client-server database systems. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 543–554, Dublin, Ireland, August 1993.

[FCL97] M. Franklin, M. Carey, and M. Livny. Transactional client-server cache consistency: Alternatives and performance. *ACM Trans. on Database Systems*, 22(3):315–363, September 1997.

[FJK96] M. Franklin, B. Jónsson, and D. Kossmann. Performance tradeoffs for client-server query processing. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 149–160, Montreal, Canada, June 1996.

[FK97] M. Franklin and D. Kossmann. Cache investment strategies. Technical Report CS-TR-3803, University of Maryland, College Park, MD 20742, May 1997. Submitted for journal publication.

[FNS91] C. Faloutsos, R. Ng, and T. Sellis. Predictive load control for flexible buffer allocation. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 265–274, Barcelona, Spain, September 1991.

[FNSY96] D. Ferguson, C. Nikolaou, J. Sairamesh, and Y. Yemini. Economic models for allocating resources in computer systems. In S. Clearwater, editor, *Market based Control of Distributed Systems*. World Scientific Press, 1996.

[FNY93] D. Ferguson, C. Nikolaou, and Y. Yemini. An economy for managing replicated data in autonomous decentralized systems. In *Proc. Int. Symposium on Autonomous and Decentralized Systems*, Kawasaki, Japan, 1993.

[Fra96] M. Franklin. *Client Data Caching: A Foundation*. Kluwer Academic Press, 1996.

[FW97] R. Fagin and E. Wimmers. Incorporating user preferences in multimedia queries. In *Proc. of the Intl. Conf. on Database Theory (ICDT)*, volume 1186 of *Lecture Notes in Computer Science (LNCS)*, pages 247–261. Springer-Verlag, January 1997.

[FYN88] D. Ferguson, Y. Yemini, and C. Nikolaou. Microeconomic algorithms for load balancing in distributed computing systems. In *Proc. of the 8th Intl. Conf. on Distributed Computing System*, 1988.

[FZ98] M. Franklin and S. Zdonik. Data in your face: Push technology in perspective. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 516–519, Seattle, WA, USA, June 1998.

[GBLP96] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total. In *Proc. IEEE Conf. on Data Engineering*, pages 152–159, New Orleans, LA, USA, 1996.

[GCGMP97] L. Gravano, C.-C. Chang, H. Garcia-Molina, and A. Paepcke. STARTS: stanford proposal for internet meta-searching. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 207–218, Tucson, AZ, USA, May 1997.

[GD87] G. Graefe and D. DeWitt. The EXODUS optimizer generator. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 160–172, San Francisco, CA, USA, May 1987.

[GGM97] L. Gravano and H. Garcia-Molina. Merging ranks from heterogeneous internet sources. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 196–205, Athens, Greece, August 1997.

[GGS96] S. Ganguly, A. Goel, and A. Silberschatz. Efficient and accurate cost models for parallel query optimizaton. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, pages 172–181, Montreal, Canada, 1996.

[GGT96] G. Gardarin, J.-R. Gruser, and Z.-H. Tang. Cost-based selection of path expression processing algorithms in object-oriented databases. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 390–401, Bombay, India, September 1996.

[GHK92] S. Ganguly, W. Hasan, and R. Krishnamurthy. Query optimization for parallel execution. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 9–18, San Diego, CA, USA, June 1992.

[GHR97] A. Gupta, V. Harinarayan, and A. Rajaraman. Virtual data technology. *ACM SIGMOD Record*, 26(4):57–61, December 1997.

[GI97] M. Garofalakis and Y. Ioannidis. Parallel query scheduling and optimization with time- and space-shared resources. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 296–305, Athens, Greece, August 1997.

[GM93] G. Graefe and W. McKenna. The Volcano optimizer generator: Extensibility and efficient search. In *Proc. IEEE Conf. on Data Engineering*, pages 209–218, Vienna, Austria, April 1993.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, San Mateo, CA, USA, 1993.

[Gra90] G. Graefe. Encapsulation of parallelism in the volcano query processing sustem. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 102–111, Atlantic City, NJ, USA, June 1990.

[Gra93] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[Gra95] G. Graefe. The cascades framework for query optimization. *IEEE Data Engeneering Bulletin*, 18(3):19–29, September 1995.

[Gra96] G. Graefe. Iterators, schedulers, and distributed-memory parallelism. *Software Practice and Experience*, 26(4):427–452, April 1996.

[GRS98] J. Goldstein, R. Ramakrishnan, and U. Shaft. Compressing relations and indexes. In *Proc. IEEE Conf. on Data Engineering*, Orlando, FL, USA, 1998.

[GS91] G. Graefe and L. Shapiro. Data compression and database performance. In *Proc. ACM/IEEE-CS Symp. on Applied Computing*, Kansas City, MO, USA, April 1991.

[GS94] J. Gwertzman and M. Seltzer. The case for geographical push-caching. Technical Report HU TR-34-94, Harvard University, Cambridge, MA, 1994.

[GW89] G. Graefe and K. Ward. Dynamic query evaluation plans. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 358–366, Portland, OR, USA, May 1989.

[HCLS97] L. Haas, M. Carey, M. Livny, and A. Shukla. Seeking the truth about *ad hoc* join costs. *The VLDB Journal*, 6(3):241–256, 1997.

[HF86] R. Hagmann and D. Ferrari. Performance analysis of several back-end database architectures. *ACM Trans. on Database Systems*, 11(1):1–26, March 1986.

[HFLP89] L. Haas, J. C. Freytag, G. Lohman, and H. Pirahesh. Extensible query processing in starburst. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 377–388, Portland, OR, USA, May 1989.

[HKWY97] L. Haas, D. Kossmann, E. Wimmers, and J. Yang. Optimizing queries across diverse data sources. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 276–285, Athens, Greece, August 1997.

[HM95] W. Hasan and R. Motwani. Coloring away communication in parallel query optimization. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 239–250, Zürich, Switzerland, September 1995.

[HMNR95] T. Härder, B. Mitschang, U. Nink, and N. Ritter. Workstation/Server-Architekturen für datenbankbasierte Ingenieuranwendungen. *Informatik – Forschung und Entwicklung*, 10(2):55–72, May 1995.

[HR96] E. Harris and K. Ramamohanarao. Join algorithm costs revisited. *The VLDB Journal*, 5(1):64–84, 1996.

[HRU96] V. Harinarayan, A. Rajaraman, and J. Ullman. Implementing data cubes efficiently. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 205–216, Montreal, Canada, June 1996.

[HS90] W. Hong and M. Stonebraker. Parallel query processing in XPRS. Technical report UCB/ERL M90/47, Department of Industrial Engineering and Operations Research and School of Business Administration, University of California, Berkeley, CA, May 1990.

[HS92] P. Haas and A. Swami. Sequential sampling procedures for query size estimation. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 341–350, June 1992.

[IC91] Y. Ioannidis and S. Christodoulakis. On the propagation of errors in the size of join results. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 268–277, Denver, CO, USA, May 1991.

[IEE98] Special issue on interoperability. IEEE Data Engineering Bulleting, Vol 21, No 3, September 1998.

[IK84] T. Ibaraki and T. Kameda. Optimal nesting for computing N-relational joins. *ACM Trans. on Database Systems*, 9(3):482–502, 1984.

[IK91] Y. Ioannidis and Y. Kang. Left-deep vs. bushy trees: An analysis of strategy spaces and its implications for query optimization. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 168–177, Denver, CO, USA, May 1991.

[INSS92] Y. Ioannidis, R. Ng, K. Shim, and T. Sellis. Parametric query optimization. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 103–114, Vancouver, Canada, August 1992.

[JWKL90] B. Jenq, D. Woelk, W. Kim, and W. Lee. Query processing in distributed ORION. In *Proc. of the Intl. Conf. on Extending Database Technology (EDBT)*, pages 169–187, Venice, Italy, March 1990.

[KB94] A. Keller and J. Basu. A predicate-based caching scheme for client-server database architectures. In *Proc. of the Intl. IEEE Conf. on Parallel and Distributed Information Systems*, pages 229–238, Austin, TX, USA, September 1994.

[KD98] N. Kabra and D. DeWitt. Efficient mid-query re-optimization for sub-optimal query execution plans. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 106–117, Seattle, WA, USA, June 1998.

[KGBW90] W. Kim, J. Garza, N. Ballou, and D. Woelk. Architecture of the ORION next-generation database system. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):109–124, March 1990.

[KGM91] T. Keller, G. Graefe, and D. Maier. Efficient assembly of complex objects. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 148–158, Denver, CO, USA, May 1991.

[kI97] 64k Incorporated. DBGuide introduction and technology overview, 1997. White paper.

[Kim94] W. Kim. *Modern Database Systems.* ACM Press/Addison-Wesley, New York, USA, 1994.

[KJA93] A. Keller, R. Jensen, and S. Agrawal. Persistence software: Bridging object-oriented programming and relational databases. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 523–528, Washington, DC, USA, May 1993.

[KK94] A. Kemper and D. Kossmann. Dual-buffering strategies in object bases. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 427–438, Santiago, Chile, September 1994.

[KKM98] A. Kemper, D. Kossmann, and F. Matthes. SAP R/3: a database application system. Tutorial handouts for the ACM SIGMOD Conference, Seattle, WA, USA, June 1998.

[KKS98] A. Kemper, D. Kossmann, and K. Stocker. Query optimization in the presence of replication. 1998. In preparation.

[KM94] A. Kemper and G. Moerkotte. *Object-Oriented Database Management: Applications in Engineering and Computer Science*. Prentice Hall, Englewood Cliffs, NJ, USA, 1994.

[KS95] R. Kimball and K. Strehlo. Why decision support fails and how to fix it. *ACM SIGMOD Record*, 24(3):92–97, September 1995.

[LA94] A. Luotonen and K. Altis. World-wide web proxies. Technical report, CERN, Geneva, Switzrland, April 1994.

[LC85] H. Lu and M. Carey. Some experimental results on distributed join algorithms in a local network. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 229–304, Stockholm, Sweden, 1985.

[LJJC98] T. Lahiri, A. Joshi, A. Jasuja, and S. Chatterjee. 50,000 users on an Oracle8 Universal Server database. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 528–530, Seattle, WA, USA, June 1998.

[LLG97] M. Leung, J. Lui, and L. Golubchik. Buffer and I/O resource pre-allocation for implementing batching and buffering techniques for video-on-demand systems. In *Proc. IEEE Conf. on Data Engineering*, pages 344–353, Birmingham, GB, 1997.

[LMSS95] A. Levy, A. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, pages 95–104, San Jose, California, USA, 1995.

[Loh88] G. Lohman. Grammar-like functional rules for representing query optimization alternatives. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 18–27, Chicago, IL, USA, May 1988.

[Lom96] D. Lomet. Replicated indexes for distributed data. In *Proc. of the Intl. IEEE Conf. on Parallel and Distributed Information Systems*, Miami Beach, FL, USA, December 1996.

[LRO96] A. Levy, A. Rajaraman, and J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 251–262, Bombay, India, September 1996.

[LRU96] A. Levy, A. Rajaraman, and J. Ullman. Answering queries using limited external query processors. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, pages 227–237, Montreal, Canada, 1996.

[LVZ93] R. Lanzelotte, P. Valduriez, and M. Zait. On the effectiveness of optimization search strategies for parallel execution spaces. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 493–504, Dublin, Ireland, August 1993.

[MAG+97] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A database management system for semistructured data. *ACM SIGMOD Record*, 26(3):54–66, September 1997.

[ME92] P. Mishra and M. Eich. Join processing in relational databases. *ACM Computing Surveys*, 24(1):63–113, March 1992.

[MGS+94] D. Maier, G. Graefe, L. Shapiro, S. Daniels, T. Keller, and B. Vance. Issues in distributed object assembly. In T. Özsu, U. Dayal, and P. Valduriez, editors, *Distributed Object Management*, pages 165–181, San Mateo, CA, USA, May 1994. Morgan Kaufmann Publishers. International Workshop on Distributed Object Management.

[ML86] L. Mackert and G. Lohman. R* optimizer validation and performance evaluation for distributed queries. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 149–159, Kyoto, Japan, 1986.

[ML89] L. Mackert and G. Lohman. Index scans using a finite LRU buffer: A validated I/O model. *ACM Trans. on Database Systems*, 14(3):401–424, September 1989.

[MLR90] T. Martin, K. Lam, and J. Russell. An evaluation of site selection algorithms for distributed query processing. *Computer Journal*, 33(1), 1990.

[MS93] J. Melton and A. Simon. *Understanding the new SQL: A complete Guide.* Morgan Kaufmann Publishers, San Mateo, CA, USA, 1993.

[NGT98] H. Naacke, G. Gardarin, and A. Tomasic. Leveraging mediator cost models with heterogeneous data sources. In *Proc. IEEE Conf. on Data Engineering*, Orlando, FL, USA, 1998.

[OS94] J. O'Toole and L. Shrira. Opportunisic log: Efficient reads in a reliable object server. Technical Report MIT/LCS-TM-506, Massachusetts Institute of Technology, Cambridge, MA 02139, March 1994.

[ÖV91] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems.* Prentice Hall, Englewood Cliffs, NJ, USA, 1991.

[Pal74] F. Palermo. A database search problem. In *Information Systems*, pages 67–101. Plenum Publ., New York, NY, 1974.

[PGGMU95] Y. Papakonstantinou, A. Gupta, H. Garcia-Molina, and J. Ullman. A query translation scheme for rapid implementation of wrappers. In *Proc. of the Conf. on Deductive and Object-Oriented Databases (DOOD)*, pages 161–186, December 1995.

[PGH96] Y. Papakonstantinou, A. Gupta, and L. Haas. Capabilities-based query rewriting in mediator systems. In *Proc. of the Intl. IEEE Conf. on Parallel and Distributed Information Systems*, Miami Beach, FL, USA, December 1996.

[PGMW95] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *Proc. IEEE Conf. on Data Engineering*, pages 251–260, Taipeh, Taiwan, 1995.

[PHH92] H. Pirahesh, J. M. Hellerstein, and W. Hasan. Extensible/rule based query rewrite optimization in starburst. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 39–48, San Diego, CA, USA, June 1992.

[PI97] V. Poosala and Y. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 486–495, Athens, Greece, August 1997.

[PIHS96] V. Poosala, Y. Ioannidis, P. Haas, and E. Shekita. Improved histograms for selectivity estimation of range predicates. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 294–305, Montreal, Canada, June 1996.

[PLH97] H. Pirahesh, T. Cliff Leung, and W. Hasan. A rule engine for query transformation in Starburst and IBM DB2 C/S DBMS. In *Proc. IEEE Conf. on Data Engineering*, pages 391–400, Birmingham, GB, 1997.

[Qia96] X. Qian. Query folding. In *Proc. IEEE Conf. on Data Engineering*, pages 48–55, New Orleans, LA, USA, 1996.

[QW97] D. Quass and J. Widom. On-line warehouse view maintenance. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 393–404, Tucson, AZ, USA, May 1997.

[Ram97] R. Ramakrishnan. *Database Management Systems*. McGraw-Hill, Inc., New York, San Francisco, Washington, D.C., 1997.

[RCK+86] N. Roussopoulos, C. Chen, S. Kelley, A. Delis, and Y. Papkonstantinou. The ADMS project: Views R Us. *IEEE Data Engeneering Bulletin*, 9(1):4–9, March 1986.

[RHS95] G. Ray, J. Haritsa, and S. Seshadri. Database compression: A performance enhancement tool. In *Proc. of the Int. Conf. on Management of Data*, Pune, India, December 1995.

[Rou91] N. Roussopoulos. The incremental access method of view cache: Concepts, algorithms, and cost analysis. *ACM Trans. on Database Systems*, 16(3):535–563, September 1991.

[RS97] M. Tork Roth and P. Schwarz. Don't scrap it, wrap it! A wrapper architecture for legacy data sources. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 266–275, Athens, Greece, August 1997.

[RSS98] L. Relly, H. Schuldt, and H.-J. Schek. Exporting database functionality – the `concert` way. *IEEE Data Engeneering Bulletin*, 21(3):40–48, September 1998.

[SAB+96] J. Sidell, P. Aoki, S. Barr, A. Sah, C. Staelin, M. Stonebraker, and A. Yu. Data replication in Mariposa. In *Proc. IEEE Conf. on Data Engineering*, pages 485–494, New Orleans, LA, USA, 1996.

[SAC+79] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access path selection in a relational database management system. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 23–34, Boston, USA, May 1979.

[SAD+94a] M.-C. Shan, R. Ahmed, J. Davis, W. Du, and W. Kent. Pegasus: A heterogeneous information management system. In W. Kim, editor, *Modern Database Systems*, chapter 32. ACM Press (Addison-Wesley publishers), Reading, MA, USA, 1994.

[SAD+94b] M. Stonebraker, P. Aoki, R. Devine, W. Litwin, and M. Olson. Mariposa: A new architecture for distributed data. In *Proc. IEEE Conf. on Data Engineering*, pages 54–65, Houston, TX, USA, 1994.

[SAL+96] M. Stonebraker, P. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu. Mariposa: A wide-area distributed database system. *The VLDB Journal*, 5(1):48–63, January 1996.

[SC90] E. Shekita and M. Carey. A performance evaluation of pointer-based joins. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 300–311, Atlantic City, NJ, May 1990.

[SC92] V. Srinivansan and M. Carey. Compensation-based on-line query processing. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 331–340, San Diego, CA, USA, June 1992.

[SD90] D. Schneider and D. DeWitt. Tradeoffs in processing complex join queries via hashing in multiprocessor database machines. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 469–480, Brisbane, Australia, August 1990.

[Sel88] T. Sellis. Multiple-query optimization. *ACM Trans. on Database Systems*, 13(1):23–52, March 1988.

[Sha86] L. Shapiro. Join processing in database systems with large main memories. *ACM Trans. on Database Systems*, 11(9):239–264, September 1986.

[SHP+96] P. Seshadri, J. Hellerstein, H. Pirahesh, T. Cliff Leung, R. Ramakrishnan, D. Srivastava, P. Stuckey, and S. Sudarshan. Cost-based optimization for magic: Algebra and implemntation. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 435–446, Montreal, Canada, June 1996.

[SK98] K. Stocker and D. Kossmann. An evaluation of greedy query optimization algorithms for distributed database systems. 1998. In prepartion.

[SKS97] A. Silberschatz, H. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill, Inc., New York, San Francisco, Washington, D.C., third edition, 1997.

[SL90] A. Sheth and J. Larson. Federated database systems for managing distributed, heterogeneous, and autonmous databases. *ACM Computing Surveys*, 22(3):183–236, September 1990.

[SM97] W. Scheufele and G. Moerkotte. On the complexity of generating optimal plans with cross products. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, pages 238–248, Tucson, AZ, USA, May 1997.

[SMK97] M. Steinbrunn, G. Moerkotte, and A. Kemper. Heuristic and randomized optimization for the join ordering problem. *The VLDB Journal*, 6(3):191–208, August 1997.

[SSV96] P. Scheuermann, J. Shim, and R. Vingralek. WATCHMAN: a data warehouse intelligent cache manager. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 51–62, Bombay, India, September 1996.

[Sto85] M. Stonebraker. The design and implementation of distributed INGRES. Addison-Wesley, Reading, MA, USA, 1985.

[Sto86] M. Stonebraker. The case for shared nothing. *IEEE Data Engeneering Bulletin*, 9(1):4–9, March 1986.

[Sto94] M. Stonebraker. *Readings in Database Systems*. Morgan Kaufmann Publishers, San Mateo, CA, USA, second edition, 1994.

[SYT93] E. Shekita, H. Young, and K.-L. Tan. Multi-join optimization for symmetric multiprocessors. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 479–492, Dublin, Ireland, August 1993.

[Tan92] A. Tanenbaum. *Modern Operating Systems*. Prentice Hall, Englewood Cliffs, NJ, USA, 1992.

[TGHM95] J. Thomas, T. Gerbes, T. Härder, and B. Mitschang. Implementing dynamic code assembly for client-based query processing. In *Proc. of the Int. Symp. for Advanced Applications, (DASFAA)*, pages 264–272, Singapore, April 1995.

[TRV96a] A. Tomasic, L. Raschid, and P. Valduriez. A data model and query processing techniques for scaling acccess to distributed heterogeneous databases in DISCO, 1996. Submitted for publication.

[TRV96b] A. Tomasic, L. Raschid, and P. Valduriez. Scaling heterogeneous databases and the design of DISCO. In *Proc. of the Intl. Conf. on Distributed Computing Systems*, 1996.

[UFA98] T. Urhan, M. Franklin, and L. Amsaleg. Cost based query scrambling for initial delays. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 130–141, Seattle, WA, USA, June 1998.

[Ull88] J. Ullman. *Principles of Data and Knowledge-Base Systems*, volume I. Computer Science Press, Woodland Hills, CA, 1988.

[VG84] P. Valduriez and G. Gardarin. Join and semijoin algorithms for a multiprocessor database machine. *ACM Trans. on Database Systems*, 9(1):133–161, March 1984.

[WDH+81] R. Williams, D. Daniels, L. Haas, G. Lapis, B. Lindsay, P. Ng, R. Obermarck, P. Selinger, A. Walker, P. Wilms, and R. Yost. R*: An overview of the architecture. IBM Research, San Jose, CA, RJ3325, December 1981. Reprinted in: M. Stonebraker (ed.), Readings in Database Systems, Morgan Kaufmann Publishers, 1994, pp. 515–536.

[WHH+92] C. Waldspurger, T. Hogg, B. Hubermann, J. Kephard, and W. Stornetta. Spawn: A distributed computational economy. *IEEE Trans. Software Eng.*, 18(2):103 – 117, Feb 1992.

[Wid95] J. Widom. Research problems in data warehousing. In *Proc. of the Intl. Conf. on Information and Knowledge Management*, pages 25–30, Baltimore, MD, USA, November 1995.

[Wie93] G. Wiederhold. Intelligent integration of information. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 434–437, Washington, DC, USA, May 1993.

[WJH97] O. Wolfson, S. Jajodia, and Y. Huang. An adaptive data replication algorithm. *ACM Trans. on Database Systems*, 22(42):255–314, June 1997.

[WKHM98] T. Westmann, D. Kossmann, S. Helmer, and G. Moerkotte. The implementation and performance of compressed databases. 1998. Submitted for publication.

[YC84] C. Yu and C. Chang. Distributed query processing. *ACM Computing Surveys*, 16(4):399–433, December 1984.

[YC91] P. Yu and D. Cornell. Buffer management based on return on consumption in a multi-query environment. *The VLDB Journal*, 2(1):1–37, January 1991.

[YKL97] J. Yang, K. Karlapalem, and Q. Li. Algorithms for materialized view design in data warehousing environment. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 136–145, Athens, Greece, August 1997.

[YL94] W. Yan and P. Larson. Performing group-by before join. In *Proc. IEEE Conf. on Data Engineering*, pages 89–100, Houston, TX, USA, 1994.

[YM97] C. Yu and W. Meng. *Principles of Database Query Processing for Advanced Applications*. Morgan Kaufmann Publishers, San Mateo, CA, USA, 1997.

[ZC97] M. Zaharioudakis and M. Carey. Highly concurrent cache consistency for indices in client-server database systems. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 50–61, Tucson, AZ, USA, May 1997.

[ZDNS98] Y. Zhao, P. Deshpande, J. Naughton, and A. Shukla. Simultaneous optimization and evaluation of multiple dimensional queries. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 271–282, Seattle, WA, USA, June 1998.

[ZL94] Q. Zhu and P. Larson. A query sampling method of estimating local cost parameters in a multidatabase system. In *Proc. IEEE Conf. on Data Engineering*, pages 144–153, Houston, TX, USA, 1994.

[ZM90] S. Zdonik and D. Maier, editors. *Readings in Object-Oriented Databases*. Morgan Kaufmann Publishers, San Mateo, CA, USA, 1990.

Contents