A Study of Query Execution Strategies for Client-Server Database Systems*

Donald Kossmann Michael J. Franklin

Department of Computer Science and UMIACS
University of Maryland
College Park, MD 20742
{ *kossmann* | *franklin* }*@cs.umd.edu*

Technical Report CS-TR-3512 and UMIACS-TR-95-85

Abstract

Query processing in a client-server database system raises the question of *where* to execute queries to minimize the communication costs and response time of a query, and to load-balance the system. This paper evaluates the two common query execution strategies, *data shipping* and *query shipping*, and a policy referred to as *hybrid shipping*. Data shipping determines that queries be executed at clients; query shipping determines that queries be executed at servers; and hybrid shipping provides the flexibility to execute queries at clients and servers. The experiments with a client-server model confirm that the query execution policy is critical for the performance of a system. Neither data nor query shipping are optimal in all situations, and the performance penalities can be substantial. Hybrid shipping at least matches the best performance of data and query shipping and shows better performance than both in many cases. The performance of hybrid shipping plans, however, is shown to be sensitive to changes in the state of the system (e.g., the load of machines and the contents of caches). Initial experiments indicate that an extended version of a 2-step optimization may be an effective strategy for adjusting plans according to the state of the system at runtime.

1 Introduction

There has been a growing realization that the needs of large classes of applications are not entirely met by the current generation of relational and object-oriented database systems. Relational database systems excel at providing high-level associative (i.e., query-based) access to large sets of flat records. In contrast, object-oriented database systems provide more powerful data modeling capabilities and are designed to support efficient navigation-based access to data. Because of these different (and in some ways, complimentary) strengths, it has become apparent that database systems combining the best aspects of the relational and object-oriented approaches are likely to gain acceptance across a larger range of applications [S+90].

System builders have been approaching this perceived need in several ways. Vendors of relational systems are moving towards integrating object-oriented features into their systems (e.g., the emerging SQL3 standard [Kul94]), and vendors of object-oriented systems are adding more powerful query facilities [Cat94].

*This work was partially supported by NSF Grant IRI-94-09575. Donald Kossmann was supported by the Humboldt-Stiftung.

Furthermore, a new class of hybrid "Object-Relational" systems has recently started to emerge (e.g., Illustra [Sto93], and UniSQL [Kim93]). These efforts have resulted in significant progress towards integrating object and relational concepts, but this progress has been primarily at the language and data model levels. In contrast, relatively little attention has been paid to the development of underlying system architectures that can efficiently support these hybrid models—particularly in a distributed environment.

The above approaches to merging relational and object technology are based on different philosophies, but they all are intended to execute in a client-server environment. The way in which particular systems use the client and server resources, however, is typically a by-product of the starting point from which the system was designed. Relational systems and their descendants are typically based on a *query shipping* policy, in which the majority of query execution work is performed at servers. Query shipping has potential benefits such as the ability to reduce communication costs for high selectivity queries and to allow for lightweight (i.e,. low cost) client machines. Furthermore, query shipping provides a relatively easy migration path from existing single-site systems, as the architecture of the database engine remains largely unchanged. Object-oriented database systems, on the other hand, are typically based on *data shipping* in which required data is faulted in to the client and processed there. Data-shipping has the advantages of exploiting the client resources (CPU, memory, and disk), reducing communication in the presence of large query results, and allowing for lighter-weight interaction between applications and the database system.

As the object and relational camps continue to merge, it becomes apparent that the dichotomy must be resolved not only at the logical levels of the system, but at the lower, architectural levels as well. As an initial step in this direction, this paper presents a study of the query and data shipping approaches for scheduling and executing queries. In terms of overall *cost* (i.e., resource usage) the trade-offs between these two approaches are fairly straightforward. However, when considering *response time*, in which parallelism can play a major role, the trade-offs are different. Furthermore, data shipping and query shipping are actually extreme points in the space of possible execution strategies. At the cost of expanding the complexity of query optimization (due to an expanded search space), a more flexible *hybrid* approach can be pursued. In this study, these three strategies are compared, with a focus on their ability to exploit parallelism between clients and servers, and among groups of servers in a larger network; their interaction with features of the client-server environment, such as dynamic client caching; and their sensitivity to changes in the run-time state of the system.

The remainder of this paper is organized as follows: Section 2 shows the options of query processing in a client/server system. Section 3 defines the data, query, and hybrid-shipping policies. Section 4 describes the experimental environment, and Section 5 presents the performance experiments. Section 6 discusses related work. Section 7 contains conclusions and proposes future work.

2 Execution Plans

Query execution plans are represented as binary trees in this study: the nodes are operators and the edges specify producer-consumer relationships between the operators. The root of a plan is always a *display* operator that passes the result of a query to an application (e.g., a graphical user interface). The leaves of a plan are always *scan* operators that read the base relations of the database.

Among others, a plan must specify the following two dimensions:

Join Ordering: exploiting the commutativity and associativety of joins, the *join* operators can be ordered arbitrarily.

Site Selection: every operator (except *display*) can be executed at any site of the system.

This study is primarily intended to demonstrate the importance of good site selection. However, a good site selection can depend on the join order, and a good join order can depend on the number of servers in the system. To focus on these two dimensions, most of the options of other dimensions are not taken into account; for example, no indexes are used, and the hybrid-hash join method [Sha86] is the only join method used.

In most database systems, the join order is restricted to a left-deep tree [SAC+79]; that is, at most one of the inputs of a join is the result of another join. In this study, however, bushy trees are allowed. Bushy trees relax the above condition and therefore, allow two or more joins to be executed independently in parallel on different sites of a distributed system.

Site selection is specified by annotating each operator with the location at which the operator is to run. (Examples of query plans with annotations are shown in Section 3.) The *display* operator at the root is always carried out at the site where the query is submitted for execution; this is specified by a *client* annotation. For the other kinds of operators, several options exist. A join can be carried out at the site of the outer relation, at the site of the inner relation, or at the site of the consumer operator that processes the result of the join. Similarly, a *select*, which is a unary operator, can be carried out at the site of the producer or the consumer.

A *scan* through a relation can be carried out by the server that owns the primary copy of the relation (*primary copy* annotation).[1] A *scan* can also be carried out at a client (*client* annotation). To execute a *scan* at a client, it is not necessary that the whole relation be cached at the client—it is possible that only portions of the relations or even no tuples at all are cached at the client. If a *scan* runs at a client, all the cached pages (in main memory or on a local disk) that contain tuples of the relations are used, and all the other pages of the relation are faulted in from the server that owns the primary copy.

At execution time, first the locations of the *display* and *scan* operators are resolved; then, the locations of the other operators are resolved given their *producer* or *consumer* annotations. Of course, the site of a producer can coincide with the site of the consumer; nevertheless, *consumer* and *producer* annotations of operators are distinguished in a plan, since relations can migrate in a distributed system and the same query can be submitted at different client machines.

[1] If horizontal partioning is used, a *scan* operator must be defined for every fragment of the relation. Partioning, however, is not taken into account in this study; an entire relation is the unit of a *scan*.

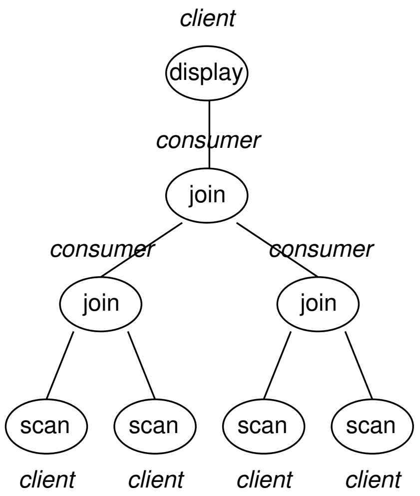


Figure 1: Example Data Shipping Plan

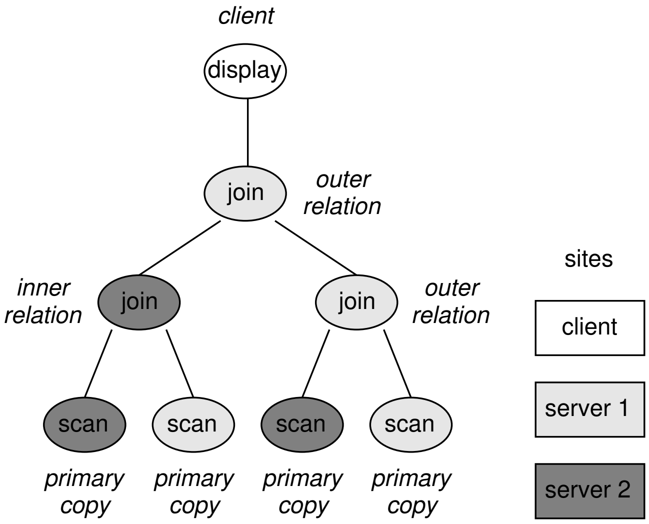


Figure 2: Example Query Shipping Plan

3 Execution Policies

In this section, data shipping, query shipping and hybrid shipping, the three query execution policies covered in this study, are described. An execution policy specifies the options allowed for site selection. A flexible policy that allows the operators of a query to be executed at different sites complicates query optimization; but, it also allows a more efficient execution of a query in a distributed system. The main characteristics of the three policies are shown in Table 1. For every operator, the possible annotations supported by the policies are listed. Operators that are not listed in Table 1 such as aggregations and projections can be annotated in the same manner as joins and selections.

	Data Shipping	Query Shipping	Hybrid Shipping
DISPLAY	client	client	client
JOIN	consumer (i.e., client)	inner relation or outer relation	consumer, inner relation or outer relation
SELECT	consumer (i.e., client)	producer	consumer or producer
SCAN	client	primary copy	client or primary copy

Table 1: Site Selection for Operators used in this Study

3.1 Data Shipping

Data shipping (DS) specifies that every operator of a query is executed at the client machine at which the query was submitted. In DS execution plans, therefore, the site annotation of every *scan* and of the *display* operator is *client*, and the annotation of all other operators is *consumer* (given that the *display* operator at the root of the tree is carried out at the client, these operators are carried out at the client as well). An example data-shipping plan is shown in Figure 1. The annotation of every operator is shown in italics, and the shading of the nodes indicates that every operator is executed at the client.

One advantage of data shipping is that it exploits the caching of data at client machines: all the data cached locally are used to evaluate a query because all the *scans* are carried out at the client. In addition, data shipping minimizes the use of the server machines, which are potential bottlenecks. DS, however, can also cause the servers to be under-utilized. Another potential disadvantage of data shipping is that it can induce unnecessary communication cost. For example, to select a few tuples of a very large relation that is not cached at the client, the whole relation is shipped from a server to the client rather than carrying out the selection at the server and sending only the few tuples that qualify.

3.2 Query Shipping

The term *query shipping* has widely been used in the context of a client-server architecture with one server machine, and in which a queries are completely evaluated at the server, with only the query result being shipped from the server to the client. There is, however, no recognized definition of query shipping for systems with many servers. For this work, we defined query shipping (QS) as the policy that places *scan* operators at the servers that own the primary copies of relations, and all the other operators (except *display*) at the site of one of their producers. For example, a *join* operator can be carried out either at the producer of the inner relation or at the producer of the outer relation. As a consequence, execution plans that support query shipping never have *consumer* annotations or *scans* that are carried out at a client machine. An example query-shipping plan is shown in Figure 2.

Obviously, query shipping can be efficient when the server machines are very powerful. In addition, query shipping provides the flexibility to load balance a system with many servers. However, query shipping does not utilize the client machines to evaluate queries; in particular, a query-shipping strategy does not exploit the caching of base relations at clients because a *scan* is always carried out at a server. This can be a serious disadvantage in systems in which server resources are limited and/or heavily loaded and can also induce extra communication cost for shipping large query results from a server to the client.

3.3 Hybrid Shipping

Hybrid shipping (HY) combines the approaches of data and query shipping. Using hybrid shipping, every operator can be annotated in any way allowed by data shipping or by query shipping. Of the three policies therefore, hybrid shipping allows the most efficient execution of a query, but it is also the most difficult policy to optimize. Figure 3 shows an example hybrid-shipping plan. As shown in Figure 3, hybrid shipping does not preclude a relation from being shipped from the client to a server (this is precluded in both data and query shipping). Shipping a relation to a server, for example, can be beneficial if the relation is cached in the client's main memory, and further processing is more efficient at the server.

To guarantee that the site of every operator can be determined at execution time from the site annotations, the optimizer must take precautions and generate *well-defined* hybrid-shipping plans. A well-defined plan has no cycles, and as a consequence, there is a path (via *producer* or *consumer* annotations) from every node of the plan to a leaf (i.e., *scan*) or to the root (i.e., *display*). A cycle, can be observed if, say, an operator A

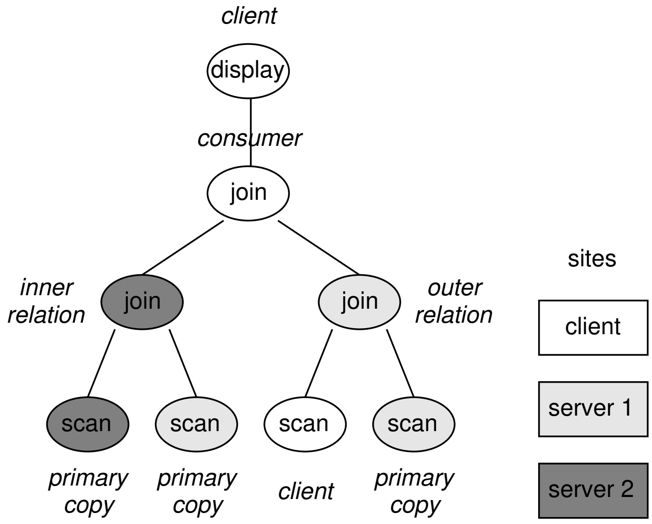


Figure 3: Example Hybrid Shipping Plan

produces the input of an operator B, and the site annotation of A is *consumer* and of B is *producer*; that is, the plan specifies that A is executed at the site of B, and B is executed at the site of A. In a tree, only such small cycles with two nodes can occur, and therefore, it is very easy to "sort out" non-well-defined plans during query optimization.

4 Experimental Environment

In order to investigate the relative performance of the data, query, and hybrid-shipping execution strategies, we developed a test environment consisting of an analytical cost model and a randomized query optimizer that is based on the model. The cost model captures the resources (CPU, disk, and network) of a group of interconnected client and server machines. It can be used to produce both total resource usage (i.e., cost) and response-time estimates. The response-time estimates are produced using the model of operator parallelism developed by Ganguly, Hasan, and Krishnamurthy [GHK92]. The query optimizer is based on randomized two-phase query optimization (2PO), which combines simulated annealing and iterative improvement, as proposed by Ioannidis and Kang [IK90]. Optimization can be aimed at minimizing either the cost or the response time predictions of the cost model. The search space explored by the optimizer includes the full range of shipping strategies; it can, however, be restricted so that the optimizer produces only data-shipping or query-shipping plans.

In this study, the query optimizer is used to generate data, query, and hybrid shipping plans for a suite of complex select-project-join queries under varying system assumptions. The quality of these plans is then assessed using the cost and response time estimates of the cost model. In some experiments the system state (i.e., resource loads and caching) at runtime is assumed to match the optimizer's expectations at compile time. For these experiments, the estimates produced during optimization are used as the performance results. Other experiments are aimed at investigating the quality of generated plans when the runtime system state differs from the expectations of the optimizer. For these experiments, the generated query plans are evaluated by re-applying the cost model with the parameters changed to reflect the new system state.

In the following, we describe the cost model, query optimizer, and the benchmark database queries. The results of the performance study are then presented in Section 5.

4.1 System Model

4.1.1 Cost Model

The analytical cost model used to control query optimization and to evaluate the quality of the resulting query plans is capable of estimating both the total cost and the response time of a query plan for a given system configuration. Following the model of Mackert and Lohman [ML86], the *cost* of a query plan is defined as the sum of the total time used to read pages from disks plus the total time used to execute CPU instructions plus the total time used to transmit messages over the network.

The *response time* estimates generated by the model are based on the approach taken in [GHK92]. The *response time* of a query is defined to be the elapsed time from the initiation of query execution until the time that the last tuple of the query result is displayed at the client. If all the operators of a plan are executed sequentially, then the response time of a query is identical to its cost. However, if parallelism (either independent or pipelined) is exploited, then the response time of a query can be lower than the total cost. Independent parallelism can arise between operators of two different sub-trees of a plan, such as scan operators on two different base relations. In contrast, pipelined parallelism arises between producer and consumer operators. Using pipelined execution, an operator can begin executing as soon as each of its producer operators has produced at least one tuple. In this case the consumer can run in parallel with its producer operators.

The model of [GHK92] estimates response times using a simplified notion of parallelism and resource contention. All operators that run in parallel are considered to complete at the same time. For independent parallelism, this response time is estimated as follows: First, the total cost (i.e., the total of all resources used) is computed for each independent operator. Thus, the model assumes that there is no overlapping of resource usage by a single operator. Second, the total usage of all resources *shared* by the independent operators 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 the operators into account. The response time of the entire group of independently parallel operators is then computed using the *maximum* of the individual cost of each operator and the *maximum* of the total usage of each shared resource. The response time of pipelined parallel operators is determined in a similar fashion, with additional consideration for the fact that only portions of pipelined operators can execute in parallel. The calculations of response time are applied to a query plan tree in a bottom-up fashion, ultimately resulting in an estimate of the response time for the entire plan.

The model of [GHK92] is intended to capture the affects of operator parallelism in a very coarse-grained fashion and is computationally efficient enough to allow many complex query plans to be evaluated in a reasonable amount of time. Although the model is not likely to accurately predict the absolute response time of a given query plan, it provides enough detail to demonstrate many of the performance implications of data, query and hybrid shipping. For these reasons, this simple model was chosen for this study.

In addition to the assumptions used in the response time model of [GHK92], the environment constructed for this study makes several other simplifications. These include:

1. Synchronization overhead between parallel operators is not modeled. For example, in pipelined parallelism, synchronization is typically required to ensure that the producer does not flood the buffers of the consumer. The cost of such synchronization is not captured in the model.

2. All joins are performed using the hybrid-hash join method [Sha86]. No indexes are used.

3. Results are obtained for single queries running in isolation. As a result, the only resource contention that is directly modeled is that resulting from the parallel execution of operators from a single query. This restriction is mitigated in two ways, however. First, memory contention is taken into account by restricting the buffer allocation given to every operator. Second, in some experiments, the operator resource demands are adjusted to simulate the load of machines induced by other queries.

4. It is assumed that all main-memory buffers are empty at the beginning of a query execution. Thus, disk I/O is always required to read the base relations from disk. Data that is cached at clients is assumed to be initially resident on the client's local disk.

4.1.2 System Parameters and Execution Model

Table 2 shows the parameters of the cost model and their default values used in the study. The parameter values are based on those used in [Fra93] and [SC90].

Parameter	Value	Description
PAGESIZE	4096	size of one data page
NOSITES	2-11	number of machines in the system
MIPS	30	CPU bandwidth of a machine
DISKTIME	20	milliseconds to read a page from disk
DISKINST	5000	CPU instructions to read a page from disk
MSGINST	20000	CPU instructions to send/receive a message
PERPAGEMI	10000	CPU instructions to send/receive 4096 bytes
NETBW	8	Mbit/s network bandwidth
DISPLAY	0	CPU instructions to display a tuple
COMPARE	2	CPU instructions to apply a predicate
HASH	9	CPU instructions to hash a tuple
MOVE	1	CPU instructions to copy 4 bytes
F	1.2	fudge factor for hybrid hash joins

Table 2: System Parameters and Default Settings

The database and the temporary relations are organized in 4KB pages (*PAGESIZE*). Pages are the unit of disk I/O and also of data transfer between sites. That is, when a producer operator is executed on a site different from its consumer operator, the producer batches its output into pages and sends a page at a time to the consumer.

A single client machine and between one and ten server machines are used in each experiment. Queries are submitted at the client machine, which has no primary copies of base relations. The client machine has a local disk which is used to cache data [FCL93] and as temporary storage during join processing. Servers are responsible for managing primary copies of relations. Each server is responsible for the primary copy of at least one base relation. The primary copy of each relation resides on a single server (i.e., relations are not declustered). In addition, a copy of a relation or a portion of it may be cached at the client machine.

The CPU and disk resources of the machines are modeled by the parameters *MIPS* and *DISKTIME* respectively. The demand on CPU resources is computed by dividing the number of instructions required for an operation by the MIPS rating. The cost of reading a page from disk is modeled as the usage of *DISKTIME* milliseconds of disk resources plus *DISKINST* instructions of CPU time. No distinction is made between random and sequential disk I/O. Although servers would be expected to have more powerful resources than clients, those resources would typically be shared among multiple clients. Therefore, in most experiments, the server and client resource parameter settings are identical. However, in some experiments, the server resource parameters are varied to simulate different loads on the system.

Communication between two sites is modeled as both CPU and network costs. CPU overhead for both senders and receivers is modeled by a fixed CPU cost per message (*MSGINST*) regardless of message size, plus an additional size-dependent component (*PERPAGEMI*). The network cost of sending a message is computed by dividing the message size by the network bandwidth (*NETBW*).

The cost of displaying the query result at the client is modeled by the *DISPLAY* parameter, which is set to zero in this study. As stated in Section 2, all joins are modeled as hybrid-hash joins. The cost of hybrid-hash joins is estimated using the cost formulas of [Sha86]; the corresponding *COMPARE, HASH, MOVE*, and F parameters are listed in Table 2. To simulate memory contention, the maximum of $\sqrt{M}$ and $\sqrt{M_B}$ buffer frames are allocated to every join[2], where M denotes the size of the inner relation multiplied by the F factor, and M_B denotes the size of a base relation (i.e., 250 pages) multiplied by the F factor. $\sqrt{M}$ Buffer frames are allocated in order to guarantee the minimum buffer allocation, and at least $\sqrt{M_B}$ buffer frames are allocated to model the execution of joins with very small relations realistically.

4.2 Query Optimization

The query plans that are evaluated in the performance study of Section 5 are obtained using randomized two-phase query optimization (2PO) [IK90]. Randomized query optimization was chosen for this study for the following reasons. First, randomized approaches have been shown to be successful at finding very good join orderings [IK90, SMK93] and generating efficient plans for parallel execution with very large search spaces [LVZ93]. Second, the simplicity of the approach allowed the optimizer to be constructed quickly, and to be easily configured to generate plans for the three different execution strategies. Third, the randomized

[2] It should be noted that limiting the buffer allocation favors data and query shipping because hybrid shipping provides the highest flexibility to exploit the aggregate main memory of the whole system.

approach optimizes very complex queries in a reasonable amount of time. For example, it takes approximately 40 seconds on a SUN Sparcstation 5 to perform join ordering and site selection for a 10-way join with 10 servers. Finally, for the purposes of this study (and in practical situations as well), it is necessary only that the generated plans be "reasonable" rather than truly *optimal*. In order to minimize the impact of randomness on the results, all of the experiments described in Section 5 were run with at least three different random seeds, and the results were averaged.

The optimizer first chooses a random plan from the desired search space (i.e., data, query, or hybrid) and then tries to improve the plan by iterative improvement (II) and simulated annealing (SA).[3] On each step, the optimizer performs one transformation of the plan. The transformations correspond to the dimensions of the search space described in Section 2. The possible moves are the following (where A, B, and C denote temporary or base relations) [4]:

1. $(A \bowtie B) \bowtie C \rightarrow A \bowtie (B \bowtie C)$
2. $(A \bowtie B) \bowtie C \rightarrow B \bowtie (A \bowtie C)$
3. $A \bowtie (B \bowtie C) \rightarrow (A \bowtie B) \bowtie C$
4. $A \bowtie (B \bowtie C) \rightarrow (A \bowtie C) \bowtie B$
5. Change the *site* field of a *join* to *consumer*, *outer relation*, or *inner relation*.
6. Change the *site* field of a *select*; i.e., either from *consumer* to *producer* or vice versa.
7. Change the *site* field of a *scan*; i.e., either form *client* to *primary copy* or vice versa.

The optimizer can be configured to generate plans from one of the three search spaces by enabling, disabling, or restricting some of the possible moves. For hybrid shipping all moves are enabled. To generate data-shipping plans, only the *join-order* moves (1 to 4) are enabled and all operators are executed at the client machine. To generate query-shipping plans, the 6th and 7th moves are disabled since all *scans* are carried out using the primary copy of a relation, and all the *selects* are executed at the same site as the corresponding *scan*. In addition, the 5th move is restricted: a *join* is never moved to the site of its consumer.

4.3 Benchmark Specification

In order to highlight the differences among the different execution strategies, the benchmark suite used in the study consists of complex queries involving 10-way joins. The large number of joins greatly expands the query plan search space, allowing more options for exploiting the resources of a distributed system with a large number of servers. In our experiments, we have found that most of the effects reported in Section 5 also arise (albeit sometimes less dramatically) when using less complex queries such as the Wisconsin Benchmark [BDT83]. However, queries with large numbers of joins are becoming increasingly common, due

[3] This study uses the same parameter settings to control the II and SA phases as used in [IK90].

[4] Note that after every move the commutativity of joins is exploited to ensure that the right (or outer) relation of each join is the larger of the two.

to applications with complex queries such as decision support and data mining, as well as to the use of path expressions in object-relational query languages.

Each relation used in the study has 10,000 tuples of 100 bytes each. In all queries, the result of a join is projected so that the size of the tuples of all temporary relations and of the query result is also 100 bytes. All joins are equi-joins and two different kinds of join graphs are used: *chain* and *star*. In a chain join graph, the relations are arranged in a linear chain and each relation except the first and the last relation is joined with exactly two other relations. In a star join graph, one center relation is joined with all of the other relations, all other relations join only with the center relation. Chain queries arise for example, when using path expressions such as *Emp.Dept.Manager.Salary*. Inter-operator parallelism can be exploited well in such queries (e.g., carrying out the two functional joins $Emp \bowtie Dept$ and $Manager \bowtie Salary$ in parallel). In contrast, independent inter-operator parallelism is difficult to exploit in a star join graph because all joins depend on the data derived from the center relation.

The benchmark contains four different chain and join queries:

SMALL: The values of the join attributes for all joins are unique (within a relation), taken from the range from 0..49,999. Since the base relations have 10,000 tuples, on an average one-fifth of the tuples of each relation qualify for the result of each two-way join. The result of a SMALL 10-way join query, therefore, is typically empty.

MEDIUM: The values of the join attributes for all joins are unique (within a relation), taken from the range of 0..9,999. The cardinality of the temporary relations and the query result in this case is 10,000 tuples—the same size as the base relations.

LARGE: The values of the join attributes for all joins are taken from the range of 0..4,999. Each value occurs twice in each relation. Therefore, the cardinality of the result of each join is twice the cardinality of each of input relations. The size of the result of the LARGE 10-way join query in this benchmark is approximately 512 MB (5 million tuples).

MIXED: The join attributes for each join are chosen randomly from among the SMALL, MEDIUM, and LARGE, join attributes.

5 Performance Experiments and Results

In this section, we examine the cost and response time trade-offs of the three shipping policies. In the experiments that follow, the query selectivities, join graphs, number of servers, client caching, and system loads are varied in order to study issues such as scalability to larger distributed systems and the robustness of compiled plans to a varying system state. In order to minimize the impact of randomized query optimization and random placement of primary base relation copies on servers, all of the data points reported here are the average of at least three different trials of the experiment. In order to compare the shipping policies, the same placements of primary base relation copies were used for all three policies for each data point.

5.1 Experiment 1: Communication Cost

The first experiment examines the cost of query execution using the three execution strategies. For these experiments, the query optimizer was configured to minimize the total query cost rather than response time. In the model used for these experiments, the costs of processing a query under the three strategies are the same except for the costs associated with communication. For this reason, this section focuses on communication requirements.

In general, the amount of communication that is required to execute a given query is determined by the following parameters:

- the sizes of the base relations
- the sizes of the intermediate relations and the query result
- the amount of relevant data cached at the client
- the number of servers and the location of the base relations

The contribution of communication to the total cost of query execution is also dependent on the relative capacities of the network (i.e., bandwidth) and the other resources in the system. Furthermore, in addition to the network cost, messages also require CPU resources at both the sending and receiving sites.

When all base relations are stored on a single server, the communication trade-offs are fairly intuitive. Query-shipping (QS) requires less communication than data-shipping (DS) when the query results are small compared to the size of the base relations, and when little or no data is cached at the client. These effects can be seen in Table 3, which shows the number of pages sent through the network to execute the SMALL, MEDIUM, and LARGE queries in a system with a single server and no client caching.[5] In the table, it can be seen that DS sends the same number of pages regardless of the selectivity. This is because all base relations are sent to the client, and the joins are performed there. In contrast, the number of pages sent by QS increases dramatically with the size of the query result: for the SMALL query, for example, the query result is empty, and no pages are shipped from the server to the client (only control messages which are not reported in Table 3 are sent). Finally, Table 3 shows that in this experiment, the communication cost of the hybrid-shipping (HY) approach is equal to the minimum of the two "pure" approaches for all three queries. It should be noted that in these experiments, the network itself (i.e., time-on-the-wire) is seldom the major factor in the total cost. In the most extreme case (the LARGE query with QS) network time accounts for 56% of the total cost. In all other cases, however, it is less than 15%.

When cached data is introduced (not shown in Table 3), the DS and HY approaches are able to exploit this data to reduce communication requirements, while the pure QS policy is not. For example, when 95% of each relation is cached at the client, DS sends only 122 pages for each query. In this situation, HY ignores the client's cached data for the SMALL query but exploits it for the other two.

[5] In this experiment, due to the single server and absence of caching, the communication requirements for chain and star queries are identical.

	SMALL Query	MEDIUM Query	LARGE Query
DS	2,441	2,441	2,441
QS	0	244	125,000
HY	0	244	2,441

Table 3: Communication Overhead (in pages)
Single Server, No Relations Cached

The communication trade-offs are somewhat different when the base relations are distributed across multiple servers. In particular, the communication required by QS and HY becomes dependent on the location of base relations and can increase as relations are distributed across more servers. If two relations that are to be joined are stored on different servers, then one of the relations must be sent to the other in order to perform the join. DS, in contrast, performs all joins at the client regardless of where the base relations are stored, and thus, its communication requirements are independent of the number of servers. These effects are demonstrated in Figures 4 and 5, which show the network traffic required to execute MEDIUM star and chain queries in a system with a varying number of servers and with 5 randomly chosen base relations (out of 10) cached at the client.

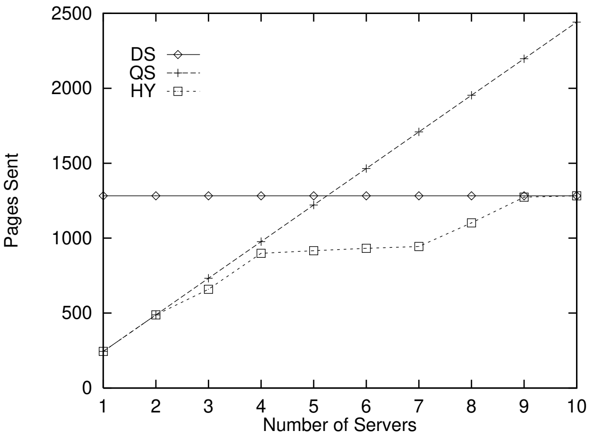


Figure 4: Pages Sent, MEDIUM Star
Varying Servers, 5 Relations Cached

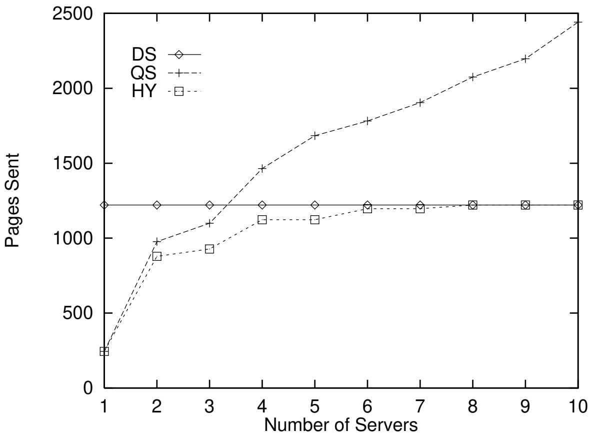


Figure 5: Pages Sent, MEDIUM Chain
Varying Servers, 5 Relations Cached

In both figures, it can be seen that the communication requirements for DS are independent of both the number of servers and the structure of the join graph. In all cases, the relations that are not cached at the client are sent to the client during query execution. In contrast to DS, the communication requirements of the other strategies are impacted by both the number of servers and the join graph structure. For the star query, the communication requirements of QS increase linearly with the number of servers (Figure 4). This is because in the chosen plans, all base relations that are located at the server containing the center relation are first joined, then the resulting intermediate relation is shipped to another server to be joined with all the relations located on that server and so on. In such a plan, one intermediate result is sent to each server that contains a non-center relation. This results in a linear increase in pages sent, because all temporary results in the MEDIUM query have the same size (they are the same size as the base relations). The communication requirements of QS for the chain query (Figure 5) also increase with the number of servers and are higher than for the star query when there are 2 to 8 servers. This behavior occurs because as stated earlier, the base relations are placed at servers randomly in these experiments, and in many cases the relations stored at a particular server are not joinable for the chain query. Thus, a server can be sent two or more base relations and/or intermediate results during the query execution.

Turning to the HY results, it can be seen that for both queries, HY always at least matches the better of the two "pure" policies and in some cases, it even has lower requirements than both of those policies. The advantage of hybrid shipping is that it can use both the primary copy of a relation to carry out joins at a server as well as a cached copy of it (if one exists) to carry out joins at the client. This additional flexibility allows HY to send fewer base relations than DS, and fewer intermediate results than QS in some cases.

Although the fundamental trade-offs are the same for the other queries (SMALL and LARGE), the relative performance of the three strategies differs due to the difference in sizes of intermediate results. For the SMALL query, which has very small intermediate results and an empty query result, QS sends fewer than 25% of the pages sent by DS for all server populations; HY behaves like QS in this case. For the LARGE query, which has relatively huge intermediate results, the communication requirements for QS are as much as two orders of magnitude greater than for DS, and HY behaves like DS in this case.

5.2 Experiment 2: Exploiting Parallelism

In the previous section, it was shown that the pure data and query-shipping strategies can both perform unnecessary communication in some situations, resulting in excessive cost. In this section, the response time of the three execution strategies is analyzed. For these experiments, the query optimizer was configured to minimize the response time estimates of the produced query plans rather than their cost (as in the previous section). In these experiments, the disks were typically found to be the most important resources in contributing to the overall response time. This is due to the assumption that main-memory buffers are empty at the beginning of query execution and because main-memory allocation is restricted. The experiments, however, show that the two pure policies (QS and DS) fully exploit neither the multiple disks nor the multiple CPUs of the system, and thus the results demonstrated in this section also apply to CPU-intensive workloads.

5.2.1 No Client Caching

The response times for the three execution strategies with varying numbers of servers and no relations cached at the client are shown for the MEDIUM star and chain queries in Figures 6 and 7 respectively. Comparing the two figures, it can be seen that the trends for the two join query graphs are similar, although the differences between the three approaches are more pronounced in the chain query results. This is because parallelism (among the server and client sites) plays a large role in determining response time and, as stated previously, the chain query has more potential for parallelism than the star query. In general, the response time of DS can be seen to be independent of both the number of servers and the shape of the query graph in this experiment. With all relations at a single server, QS performs worse than DS here, but as relations are spread across servers, QS better exploits the additional parallelism and thus, has better performance than DS for two or more servers. The trade-off here is that DS exploits the client resources, while QS exploits the servers' resources. As more servers are added, the client resources are dominated by the server resources, and thus QS performs better than DS. HY again has the best performance throughout the range of server populations; beating both pure strategies when few servers are used, because of its ability to use the client resources for *load balancing*, and matching the performance of QS when a larger number of servers are used.

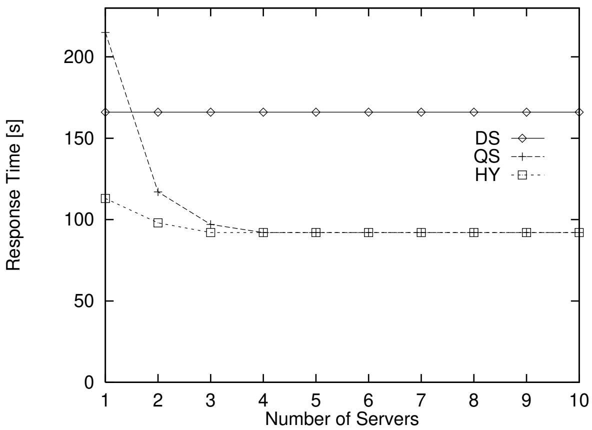


Figure 6: Response Time (secs), MEDIUM Star
Varying Servers, No Caching

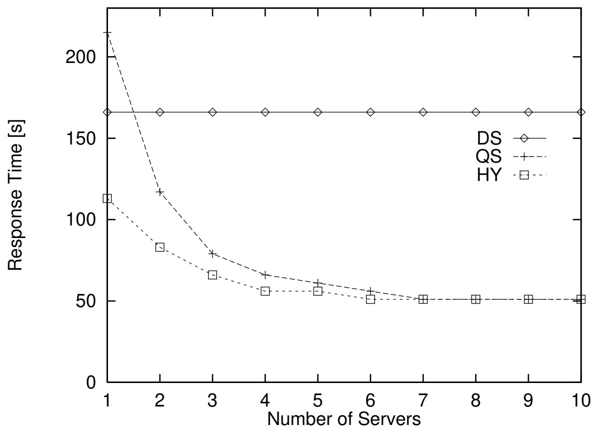


Figure 7: Response Time (secs), MEDIUM Chain
Varying Servers, No Caching

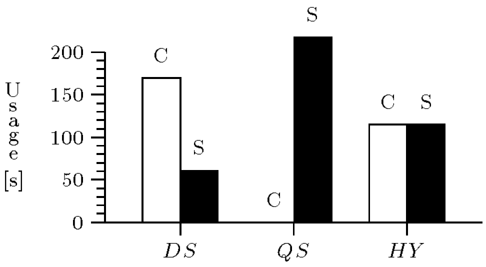


Figure 8: CPU + Disk Usage, MEDIUM Chain
1 Client (C), 1 Server (S), No Caching

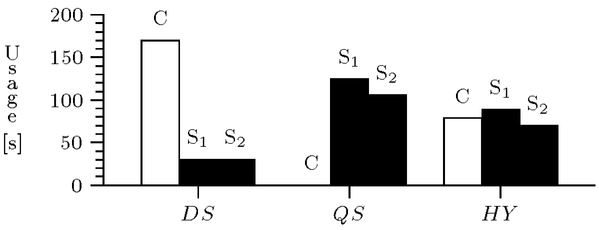


Figure 9: CPU + Disk Usage, MEDIUM Chain
1 Client (C), 2 Servers (S_1,S_2), No Caching

The response time results are driven in large part by the parallelism that is obtained by the three approaches. This parallelism is depicted in Figures 8 and 9, which show the sum of the CPU and disk usage at each site with one and two servers respectively for the MEDIUM chain query (whose response time was shown in Figure 7). In the one-server case, DS executes all joins at the client and faults in the pages of the base relations from the server's disks; DS can, therefore, exploit *pipelined* parallelism. QS, performs all work except for displaying the result at the single server. As a consequence, QS does not exploit parallelism and has poorer performance in this case. HY, in contrast, is able to balance the load between the client and the server, thus obtaining the best response time here. In the two-server case (Figure 9), the DS response time is still dominated by the client's usage, which remains unchanged from the single server case. QS, however, is able to exploit independent parallelism between the two servers and has a significant improvement in response time. Finally, HY is again able to better exploit the resources of all of the sites resulting in a better response time than the two pure approaches.

5.2.2 Impact of Client Disk Caching

The results of the previous section demonstrate that when only small numbers of servers are involved in a query, the client resources can be exploited to provide a substantial improvement in response time by aiding in the processing of joins. An additional way that clients can improve query performance when server resources are limited is by performing *scans* of cached data. Figures 10 and 11 show the impact of client disk caching on the response time of the three different types of plans for the SMALL and MEDIUM chain queries respectively, when all relations are stored on a single server, and the number of relations cached at the client is varied from zero to ten.

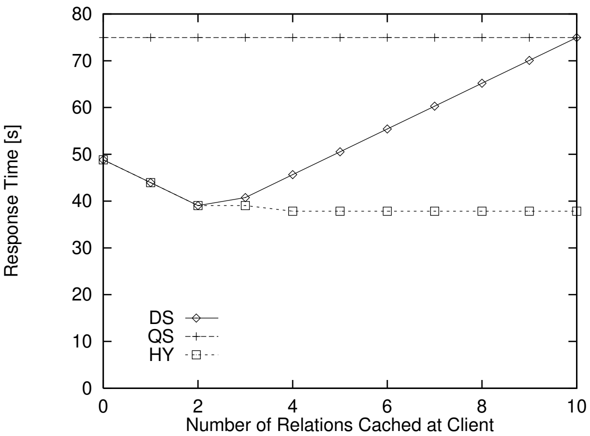


Figure 10: Response Time (secs), SMALL Chain
Single Server, Varying Caching

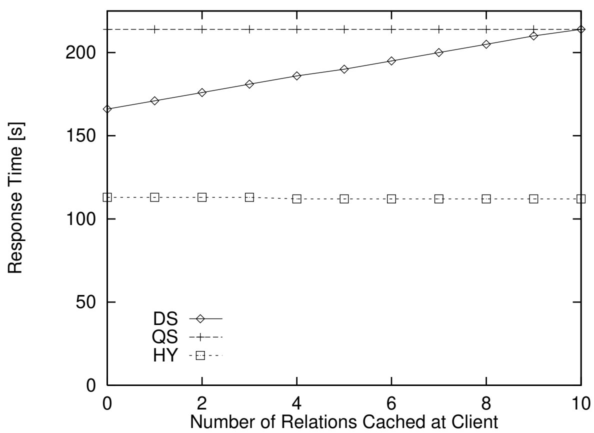


Figure 11: Response Time (secs), MEDIUM Chain
Single Server, Varying Caching

The response time of QS is independent of the number of relations cached for both queries. This is because QS always performs all scans and joins at the server, thereby ignoring any cached copies. In contrast, the DS response time (and to a lesser extent, the HY response time) is affected by the number of cached relations. Turning to the SMALL chain query results (Figure 10), it can be seen that the response time of DS improves when the number of cached relations is increased from 0 to 2, but then degrades from that point on. Recall that DS performs all joins and scans at the client, and therefore, if relations are cached on the client, then the client's disk is used for both scanning relations and for temporary storage for hybrid-hash joins. When a small number of relations are cached, DS can exploit pipelined parallelism between the server and the client. However, as more relations are cached, DS results in increased load on the client disk and decreased load on the server disk. For this reason, the performance of the DS plans degrades beyond the caching of two relations. When all ten relations are cached at the client, all scan and join I/O is performed at a single site (i.e., the client) and thus, the performance of DS and QS converge. As can be seen in the figure, the HY approach uses the proper amount of cached data in this case; it matches the performance of DS when up to two relations are cached, and then stabilizes at that point, while the performance of DS begins to degrade.

These results demonstrate that while client caching has the potential to improve query performance, the best performance can sometimes be achieved by ignoring some cached data.

Figure 11 shows the results for the MEDIUM chain query. In this case, the performance of the DS plan is harmed by any caching of relations. This effect differs from what occurs in the SMALL case because of the amount of work that must be done for joins here. For the SMALL query, the high selectivity of the joins results in small inner relations, so that most of the joins are processed in a single pass of the hybrid-hash join algorithm. In contrast, the join results in the MEDIUM query are the same size as the base relations. Since MEDIUM queries are given the minimum memory allocation, all joins require a number of passes through the hybrid-hash algorithm, resulting in a much higher I/O requirement than in the SMALL case. Since DS performs all joins at the client, the scans of cached relations interfere with the I/O required for joins, resulting in the degradation of performance demonstrated in Figure 11. In this case, the HY approach ignores all cached copies and uses the client disk solely for join processing, resulting in better performance than both DS and QS.

5.3 Robustness of Compiled Plans

The previous sections have shown that the hybrid-shipping approach can have benefits in both cost and response time because it is effective at exploiting the resources of both the client and the servers. The power of hybrid shipping comes from its ability to produce a query plan that is tailored to a particular system configuration. This section investigates the robustness of hybrid-shipping plans when the state of the system at runtime differs from what was expected by the query optimizer. Two issues are addressed in this section: 1) changes to the load on the server, and 2) changes to the state of the client cache. In general, the results show that as expected, the performance of a given hybrid-shipping plan can degrade significantly if the runtime state differs from what was expected at compile-time. This demonstrates the need for dynamic approaches to developing (or adjusting) query plans, which will be addressed in Section 5.4.

5.3.1 Server Load

In this section, we study the performance of compiled plans when the load on the server is varied with respect to what was expected at optimization-time. In this experiment, the load on the server is simulated by varying the settings of the *MIPS* and *DISKTIME* parameters (described in Table 2) compared to what is used by the cost model when the optimizer is run. In this experiment, a hybrid-shipping query plan that is generated assuming the default settings for the server resources ($MIPS = 30$, $DISKTIME = 20$ ms) is executed on a system where the server resources are varied from 1/8 of the default speeds ($MIPS = 3.75$, $DISKTIME = 160$ ms) to 8 times the default speeds ($MIPS = 240$, $DISKTIME = 2.5$ ms). Figure 12 shows the results of this experiment for the MEDIUM chain query with a single server and no client caching. On the x-axis is the speed of the server resource at runtime relative to what is assumed by the optimizer, and on the y-axis is the response time of the pre-compiled hybrid-shipping plan (compHY) relative to the response time of a plan optimized with the run-time resource settings.

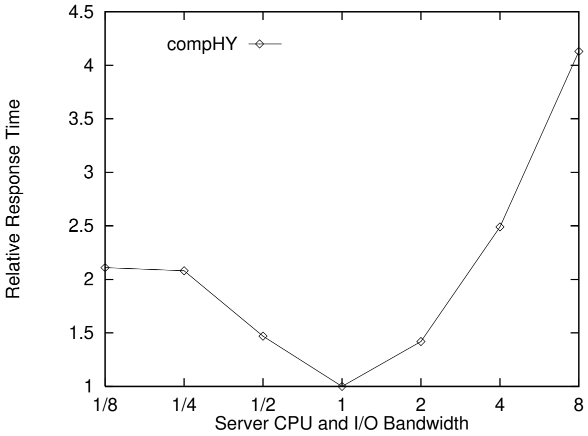


Figure 12: *Relative* Response Time, MEDIUM Chain
Single Server, No Caching

At an x-axis value of 1, the compile-time assumptions and the run-time settings are the same, so the relative response time equals one. Moving left along the x-axis, the server becomes slower (i.e., more heavily loaded) relative to the optimizer settings. In this case, the compiled plan relies more heavily on server resources than it should, resulting in poorer performance. In this case, the degradation in performance plateaus at slightly worse than a factor of 2 slower than the best case. Moving to the right, the server becomes faster (i.e., less heavily loaded) relative to the optimizer settings. The relative performance of the pre-compiled plan degrades dramatically in this case, being more than a factor of four slower than the best case. When the optimizer underestimates the server resources, it places a large number of joins on the client (8 of the 9 joins in these experiments), which results in much lower performance than could be obtained using the fast server resources to execute all the joins on the server.

5.3.2 Client Disk Caching

Another significant way that the run-time state of the system can differ from what is assumed at query optimization time is the contents of client disk caches. Because caching is inherently a dynamic process, it is difficult to predict what will be in a particular client's cache at any given time. As described in Section 5.2.2, client disk caching can play an important role in determining query performance when the server resources are limited and when relation scanning is a substantial part of the cost of query execution. For this reason, this experiment examines the dynamic impact of caching for the SMALL chain query with a single server. Table 4 shows the response time of pre-compiled hybrid-shipping plans (compHY) assuming four different client cache states when they are executed against systems with those four cache states at run-time. The last column shows the best case for this experiment, namely, when the run-time state matches the assumption made by the optimizer. The cache contents are represented by (x, y) pairs where x represents the number of relations cached, and y represents the percentage of each of those relations that are cached. The plan "compHY(10,50%)" is a hybrid-shipping plan that is compiled assuming the client cache contains 50% of

Run-time Cache Contents (# Relations, Portion)	Compiled Plans compHY (0, 0%)	compHY (10, 50%)	compHY (5, 100%)	compHY (10, 100%)	Best Case
(0, 0%)	48.83	56.64	56.64	60.14	48.83
(10, 50%)	48.83	37.84	46.88	48.75	37.84
(5, 100%)	48.83	37.84	37.84	48.75	37.84
(10, 100%)	48.83	57.37	37.84	37.84	37.84

Table 4: Response Time (secs), SMALL Chain
Single Server, Expected vs. Actual (Run-Time) Caching

the tuples from each of the ten relations.

In this experiment, the hybrid-shipping plans are sensitive to the client caching state, but less so than to the server load (shown in Section 5.3.1). The worst case in this experiment is a response time increase of 59% compared to a plan with perfect knowledge of the cache contents. compHY(0,0%), which assumes an empty cache, suffers a 29% penalty compared to the best case in all the three of the cases where the cache is not empty at run-time. This is because it performs all of its scans at the server and all joins at the client. In contrast, compHY(10,100%), which assumes that all data is cached at the client, suffers a 59% penalty in the case where the client cache is empty at run time and a 29% penalty in the other cases. This sensitivity to an empty cache results from the fact that the plan executes many joins on the server. These joins must compete for server resources with the I/O required to read the non-cached data. The compHY(10,50%) plan pays a high performance penalty when all relations are fully cached: the optimizer places all the scans at the client and, in addition, places several joins at the client assuming that half of the pages for the scans must be read from the server's disk. At execution time, high contention on the client's local disk can be observed because (unlike the expectation) all the pages of the 10 relations are read from the client's local disk. The most robust plan in this experiment is compHY(5,100%), which assumes that five relations are cached at the client in their entirety. This plan pays a penalty of 16% when the client cache is empty, and 29% when the cache contains half of all ten relations, and is equal to the best plan in the other two cases.

5.4 2-Step Optimization

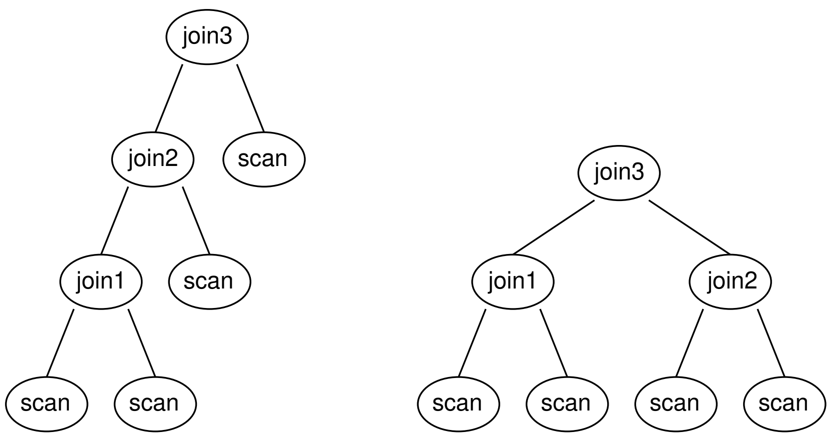
The results of the previous section show that the performance of a compiled plan can degrade significantly if the server load and/or client cache contents at run-time differ from what is assumed by the query optimizer at compile-time. The main factor contributing to the performance degradation was seen to be the site selection, rather than the join ordering. This observation indicates that a 2-step optimization similar to that proposed by Carey and Lu [CL86] and similar to that used by XPRS [HS90] and Mariposa [SAD+94] may be a good basis for generating query plans that can be modified dynamically prior to query execution in order to adapt to changes in the run-time environment. A 2-step optimizer for a client-server query-processing environment would perform the following:

1. (*Join Ordering*): At compile time, generate a good query plan assuming that the query is going to be evaluated by a centralized system with one machine only.

2. (*Site Selection*): At execution time, determine where to execute every operator of the plan and choose which cached data (if any) to use.

Join ordering can be carried out with any existing query optimizer. The experiments reported in this section use the randomized query optimizer described in Section 4.2, configured to carry out only the moves related to join ordering. Simulated annealing (SA) is used for the site-selection step at execution time. In practice, SA is somewhat slow for execution-time use (it takes approximately 8.5 seconds on a SparcStation 5, in these experiments). SA, however, is sufficient for the purpose of these experiments, which is to investigate the potential benefits and limitations of a 2-step approach for client-server query execution.

It should be clear that in general, 2-step optimization cannot always generate an optimal distributed plan. For example, if join ordering is performed assuming a single execution site, a 2-step approach may choose a left-deep join tree (shown in Figure 13a) rather than a bushy tree (shown in Figure 13b). The bushy tree might have a higher cost on a single site but a lower response time in a distributed system if *join1* and *join2* can be carried out at different sites in parallel.



(a) deep tree (b) balanced tree

Figure 13: Possible Join Orders

The performance of hybrid-shipping plans generated by a 2-step optimization process in which join ordering is done assuming a centralized system is shown for all four variants (SMALL, MEDIUM, LARGE, and MIXED) of the chain query in Figure 14. The figure shows the performance of the 2-step plans relative to that of a plan generated by a 1-step optimizer that has perfect knowledge of the placement of base relations on servers.

For the MEDIUM and MIXED queries, the 2-step plan has increasingly worse performance than the 1-step plan as servers are added. For the SMALL query, join processing is cheap because the intermediate results are small, and the performance penality of a 2-step approach is fairly small. For the LARGE query

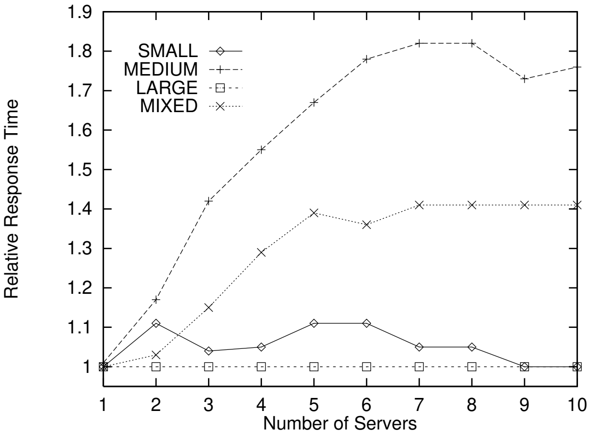


Figure 14: *Relative* Response Time, Chain Queries
2-Step with CENTRALIZED Join Ordering

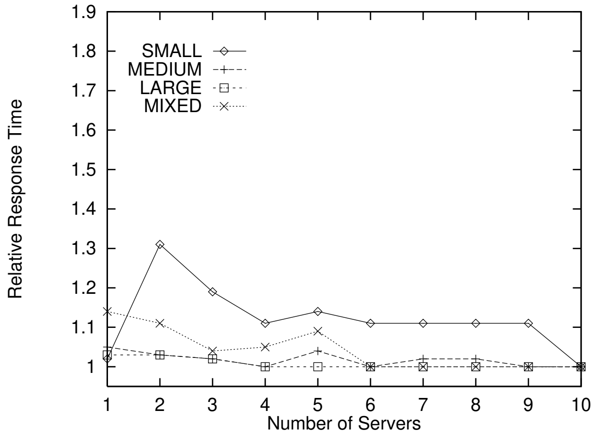


Figure 15: *Relative* Response Time, Chain Queries
2-Step with DISTRIBUTED Join Ordering

both optimizers generate a bushy tree plan because the low selectivity of the joins results in such a plan having the lowest cost even in a centralized system. In contrast, the performance of the 2-step plan for the MEDIUM query is more than 80% slower than that of the 1-step plan when many servers are used. This is because 2-step optimization chooses a deep join order in this case, while 1-step chooses a bushy tree plan. The shape of a tree for a MIXED query is less predictable because joins with different selectivities (from low to high) are found in the query. In this case, the relative performance of 2-step optimization is better than for the MEDIUM query since the joins with high selectivity are carried out first, and thus, intermediate relations are smaller than those in the MEDIUM query.

The problems encountered by 2-step optimization in Figure 14 arise largely because join ordering is performed using a centralized model. An alternative to this approach is to perform join ordering in 2-step optimization assuming that the primary copy of every relation is located at a different server (i.e., assuming that the system has 10 servers). The net effect of this assumption is that join orders are always generated as bushy trees. The performance results for this 2-step optimization are shown in Figure 15. As expected, this approach performs well for the LARGE, MEDIUM, and MIXED queries in this case, because they all do better with bushy tree join orderings. In addition the relative performance for these queries tends to improve as servers are added. In contrast, the SMALL query, which performs best with a deep join ordering, has the worst relative performance in this case.

The results of these experiments show that neither approach to 2-step optimization is likely on its own to be robust across a wide range of queries and system configurations. In this case, however, the combination of the two approaches appears to be sufficient: choosing the best of the two variants suffers at most a 14% performance penalty for the four chain queries. In this environment, 2-step optimization is even more robust for the star queries (not shown), as star queries do not parallelize as well in any case (see Figure 6). In these experiments, therefore, the star query plan generated by the best variant of 2-step optimization never has a noticeably high relative response time. For star queries, however, a pre-compiled join order can have a very high communication cost because it sometimes cannot take advantage of the fact that relations that are located at the same site should be joined locally before sending the result to another site.

A simple way to combine the two variants is to build a 2-step optimizer that generates both join orders at compile time, and then chooses the best plan (after site selection) at run-time. However, while these results show that 2-step optimization is a potential approach towards addressing the need for dynamic query optimization in the client-server environment, significant additional work is required in order to develop practical and robust implementations of such an approach.

6 Related Work

Distributed database systems were first investigated in the late seventies; e.g., [ESW78]. At that time, several prototype systems were developed such as: System R* [WDH+81], SDD-1 [BGW+81], and distributed INGRES [Sto85]. Typically, these systems focused on optimizing the cost of a query; in particular, much effort was made to minimize the communication cost since the systems were designed to run on slow networks.[6] Many of the concepts are still valid, today. In the eighties, however, the client-server paradigm emerged as the standard for any kind of distributed data processing leading to a shift in research directions. This study, therefore, uses a client-server environment. In addition to cost, this study also analyzes the response time of queries, the parallel execution of operators of a query on different sites, and the load balancing of systems.

Load balancing has also been investigated by Carey and Lu [CL86]; they propose an approach that exploits the replication of data on different sites. Even though their system model, their workloads, and their execution strategy was very different from our settings, they basically came to the same conclusion that a system should provide the flexibility to carry out a query at different sites to improve response time.

Hagmann and Ferrari were among the first to study query processing in a client-server environment [HF86]. They investigated different ways to split the functionality of a DBMS (e.g., query parsing, optimization, and execution) between client (front-end in their terminology) and server (back-end) machines. In our study, however, it was assumed that all the functions of a DBMS were available at all the sites of a system, and we concentrated on the performance of executing queries (e.g., joins) because in most applications, this part of query processing is likely to generate the most load on the system.

By now, many studies have been carried out investigating various aspects of client-server databases; e.g., [DFMV90, FC94]. Most of these studies, however, have been influenced by the dichotomy between relational and object-oriented systems and were carried out using either pure query shipping or pure data shipping. A related study that analyzed the utilization of resources in client-server systems was carried out by Delis and Roussopoulos [DR92]. Like many other studies, they concentrated on a system with only one server, and therefore, could not realize the potential to execute the operators of a query on different sites in parallel.

Examples of research prototypes that support multiple servers are Orion-2 [JWKL90], SHORE [CDF+94] and Mariposa [SAD+94, SDK+94]. In these systems, the design of the execution policy plays an important role. In the current version, SHORE only uses data shipping. Mariposa supports both query and data ship-

[6] Only distributed INGRES could be parametrized to optimize response time.

ping. Several execution policies that allow a flexible decision where to execute a query are also incorporated in Orion. An initial discussion of execution policies for the specific task of distributed object assembly can also be found in [MGS+94].

Multidatabases [SL90] are also examples of distributed systems. It should be noticed, however, that all the multidatabase systems seem to use a variant of data shipping because only a fixed set of functions can be carried out on a server in those systems, and thus, operations processing data stored in two or more—possibly heterogeneous—databases of the system must be carried out at a client. This study also varies from the performance studies that have been carried out for the design of parallel database systems. Much of the emphasis in studying parallel database systems was to exploit *intra*-operator parallelism [DG92]. This study, however, showed how *inter*-operator parallelism can be exploited in a distributed system.

7 Conclusion

In this work, three different execution policies were identified and evaluated in client-server database systems:

1. *data shipping*, used in most object-oriented database systems;
2. *query shipping*, used in most relational systems;
3. *hybrid shipping* that aims to combine the advantages of data and query shipping and to support navigation-based and query-based access to the database.

The performance experiments revealed that depending on the number of servers in the system, the caching of data at clients, and the query characteristics, the execution policy influences the performance of a system significantly. Both data and query shipping induce unnecessary communication overhead in some cases. Furthermore, the "pure" approaches do not fully utilize the resources of a system which can result in an increased response time of queries.

The best performance can be achieved by a hybrid execution policy that provides the flexibility to place the operators of a query on many sites. This approach, however, expands largely the space of possible plans to evaluate a query. It is also very sensitive to the load of the machines and to the amount of data cached on the clients' local disks. Initial experiments indicate that a dynamic site selection in an extended 2-step approach that generates several join orders at compile time might support a hybrid execution policy sufficiently.

This study was focused on rather complex queries and disk-bound computation. In future work, we intend to analyze the effects of navigation-based access, updates and the utilization of the aggregate main memory. In addition, we plan to investigate the performance of a system when several applications run concurrently rather than measuring queries in isolation. We are also implementing a query engine on top of the SHORE storage system, in order to further investigate these issues.

References

[BDT83] D. Bitton, D. J. DeWitt, and C. Turbyfil. Benchmarking database systems: A systematic approach. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, 1983.

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

[Cat94] R. G. G. Cattell. *Object Database Standard*. Morgan-Kaufmann Publ. Co., San Mateo, CA, USA, 1994.

[CDF+94] M. J. Carey, D. J. DeWitt, M. J. Franklin, N. E. Hall, M. L. McAuliffe, J. F. Naughton, D. T. Schuh, M. H. Solomon, C. K. Tan, O. G. Tsatalos, S. J. White, and M. J. Zwilling. Shoring up persistent applications. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 383–394, Minneapolis, MI, USA, May 1994.

[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.

[DFMV90] D. J. 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.

[DR92] A. Delis and N. Roussopoulos. Performance and scalability of client-server database architectures. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 610–623, Vancouver, Canada, 1992.

[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*, 1978.

[FC94] M. Franklin and M. Carey. Client-server caching revisited. In Özsu et al. [ÖDV94]. International Workshop on Distributed Object Management.

[FCL93] M. J. Franklin, M. J. 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, 1993.

[Fra93] M. Franklin. *Caching and Memory Management in Client-Server Database Systems*. PhD thesis, University of Wisconsin, Madison, Wisconsin, June 1993.

[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, USA, June 1992.

[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.

[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.

[IK90] Y. E. Ioannidis and Y. C. Kang. Randomized algorithms for optimizing large join queries. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 312–321, Atlantic City, USA, April 1990.

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

[Kim93] W. Kim. Object-oriented database systems: Promises, reality, and future. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, Dublin, Ireland, 1993.

[Kul94] K. G. Kulkarni. Object-oriented extensions in SQL3: A status report. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, page 478, Minneapolis, MI, USA, May 1994.

[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, 1993.

[MGS+94] D. Maier, G. Graefe, L. Shapiro, S. Daniels, T. Keller, and B. Vance. Issues in distributed object assembly. In Özsu et al. [ÖDV94], pages 165–181. 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.

[ÖDV94] T. Özsu, U. Dayal, and P. Valduriez, editors. *Distributed Object Management*. Morgan-Kaufmann Publ. Co., San Mateo, CA, USA, May 1994. International Workshop on Distributed Object Management.

[S+90] M. Stonebraker et al. Third generation data base system manifesto. Technical Report No. UCB/ERL M90/28, UC Berkeley, Berkeley, CA, 1990.

[SAC+79] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. 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+94] M. Stonebraker, P. M. 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, 1994.

[SC90] E. J. Shekita and M. J. 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 90.

[SDK+94] M. Stonebraker, R. Devine, M. Kornacker, W. Litwin, A. Pfeffer, A. Sah, and C. Staelin. An economic paradigm for query processing and data migration in Mariposa. In *Proc. of the IEEE Conf. on Parallel and Distributed Information Systems*, pages 58–67, September 1994.

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

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

[SMK93] M. Steinbrunn, G. Moerkotte, and A. Kemper. Optimizing join orders. Technical Report MIP-9307, Universität Passau, 94030 Passau, Germany, 1993.

[Sto85] M. Stonebraker. The design and implementation of distributed INGRES. In M. Stonebraker, editor, *The INGRES Papers: Anatomy of a Relational Database System*. Addison-Wesley Pub., 1985.

[Sto93] M. Stonebraker. The Miro DBMS. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, Washington, DC, USA, May 1993.

[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.