Performance Tradeoffs for Client-Server Query Processing*

Michael J. Franklin Björn Thór Jónsson Donald Kossmann
Department of Computer Science and Institute for Advanced Computer Studies
University of Maryland, College Park
{*franklin* | *bthj* | *kossmann*}@*cs.umd.edu*

Abstract

The construction of high-performance database systems that combine the best aspects of the relational and object-oriented approaches requires the design of client-server architectures that can fully exploit client and server resources in a flexible manner. The two predominant paradigms for client-server query execution are data-shipping and query-shipping. We first define these policies in terms of the restrictions they place on operator site selection during query optimization. We then investigate the performance tradeoffs between them for bulk query processing. While each strategy has advantages, neither one on its own is efficient across a wide range of circumstances. We describe and evaluate a more flexible policy called hybrid-shipping, which can execute queries at clients, servers, or any combination of the two. Hybrid-shipping is shown to at least match the best of the two "pure" policies, and in some situations, to perform better than both. The implementation of hybrid-shipping raises a number of difficult problems for query optimization. We describe an initial investigation into the use of a 2-step query optimization strategy as a way of addressing these issues.

1 Introduction

The needs of many classes of applications are not adequately 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 (e.g., [S$^+$90]).

1.1 Merging RDBMS and ODBMS Functionality

Database system builders have been approaching this perceived need in several ways. Relational vendors are moving towards integrating object-oriented features into their systems (e.g., the emerging SQL3 standard [Kul94]) while vendors of object-oriented systems are adding more powerful query facilities [Cat94]. Furthermore, a new class of hybrid "Object-Relational" systems has recently started to appear (e.g., Illustra and UniSQL). 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.

While most modern database systems are designed to execute in a client-server environment, relational and object-oriented systems tend to exploit the resources of such environments in significantly different ways. Relational systems and their descendants are typically based on *query shipping*, in which the majority of query processing is performed at servers. The benefits of query shipping include: the reduction of communication costs for high selectivity queries, the ability to exploit server resources when they are plentiful, and the ability to tolerate resource-poor (i.e., low cost) client machines. Object-oriented database systems, on the other hand, are typically based on *data shipping*, in which required data is faulted in to the client to be processed, and possibly cached, there. Data-shipping has the advantages of exploiting the resources (CPU, memory, and disk) of powerful client machines and reducing communication in the presence of locality or large query results. Both of these factors allow data-shipping to scale as users (and hence, their desktop resources) are added to the system. Data-shipping also allows for light-weight interaction between applications and the database system, as is needed to support navigational data access.

1.2 Architectural Implications

In order to build database systems that combine the best aspects of relational and object-oriented systems, it is essential that the dichotomy between these approaches be resolved not only at the logical levels of the system, but at the lower, architectural levels as well. As part of the DIMSUM project[1] we have been investigating the integration of the data-shipping and query-shipping models. In this paper we focus on the tradeoffs that arise between these approaches for processing bulk queries.

*This work was partially supported by NSF Grant IRI-94-09575, an IBM SUR award, and a grant from Bellcore. Björn Jónsson was supported in part by a Fulbright Fellowship. Donald Kossmann was supported in part by the Humboldt-Stiftung.

[1]Distributed Information Management Systems at the University of Maryland (`http://www.cs.umd.edu/projects/dimsum`).

The primary distinction between data-shipping and query-shipping with regard to query processing relates to the use of client resources. Data-shipping exploits client resources in two ways. First, query operators (e.g., selects, projects, joins, etc.) are executed at a client. This use of client resources is beneficial when the resources available at the client are significant relative to those at the servers. This situation can arise for example, if servers are heavily loaded. The second way that client resources are used arises only if client data caching is supported. In this case, scans of base data can also be performed at the clients. Performing scans at clients can save communication and offload server disk resources. A query-shipping policy does not exploit either of these uses, and thus may be detrimental to performance under certain conditions. Data-shipping is not a performance panacea, however. In fact, when server resources are plentiful, or if locality of data access at clients is poor, then a query-shipping approach can have significant performance advantages.

It should be obvious from the above considerations that neither data-shipping nor query-shipping is the best policy for query processing in all situations. As a result, a system that supports only one of these policies is likely to have sub-optimal performance under certain workload and/or system conditions. A potential solution to this problem is to build a system that is capable of executing queries using either approach and to allow the query optimizer to choose between the two. While such a solution goes a long way towards solving the problem, there are cases where neither the pure data-shipping nor the pure query-shipping approach can provide the best performance. In particular, for complex queries, a *hybrid* approach can in some cases outperform both pure policies.

1.3 Overview

In this study, we compare the performance of pure data-shipping, pure query-shipping, and a hybrid-shipping approach that has the flexibility to place individual query operators and scans at servers or at clients on a case-by-base basis. The flexibility of a hybrid approach allows it to potentially meet or beat the performance of both pure policies. This flexibility, however, comes at a cost of increased complexity in query optimization. The search space for hybrid plans is significantly larger than for the two restricted approaches. Furthermore, the quality of a hybrid plan is sensitive to the accuracy of the optimizer's cost model and changes to the state of the distributed query environment (e.g., in terms of load, client cache contents, data placement, etc.).

We have constructed a randomized query optimizer and a detailed simulation environment that allow us to compare the performance of query execution plans under varying workload and system conditions. We use these tools to compare the three approaches, using the hybrid approach as an "ideal" case that serves to gauge the performance of the two pure approaches. We then describe an initial investigation of a 2-step query optimization strategy as a step towards developing a robust hybrid-shipping approach.

The remainder of this paper is organized as follows: Section 2 describes the options for client-server query processing and defines the three execution policies covered in this study. Section 3 describes the experimental environment. Section 4 presents a set of experiments that investigate the tradeoffs among the three execution policies. Section 5 discusses query optimization issues for flexible client-server systems. Section 6 discusses related work. Section 7 contains conclusions and proposes future work.

2 Query Execution Policies

2.1 Execution Plans

In this section, we describe the three query execution policies covered in this study: data-shipping, query-shipping and hybrid-shipping. The study focuses on select-project-join queries.[2] Execution plans for such queries can be represented as binary trees in which the nodes are query operators and the edges represent producer-consumer relationships between the operators. A query plan specifies the ordering of operators, the placement of operators at sites, and the methods to be employed for executing each operator (e.g., the join method). The three execution polices that are covered in this study differ primarily in the placement of operators at sites — this is referred to as *site selection*.

Site selection for operators is specified by annotating each operator with the location at which the operator is to run. These annotations refer to *logical* sites, such as "client", "primary copy", "consumer", "producer", etc., and are not bound to physical machines until query execution time. The query operators and their possible annotations are as follows:

- *Display* - The root of a query plan is always a display operator. This operator presents the results of the query to the user or an application program, and thus, it is always given the annotation *client*, indicating that it must be executed at the site where the query is submitted.

- *Join*[3] - A join combines the input from two producers and generates a single output stream. A join can be given one of three annotations: *consumer*, *inner relation*, or *outer relation*. A *consumer* annotation states that the operator should be executed at the same site as the operator that consumes its output. An *inner relation* annotation indicates that the operator should be executed at the same site as the operator that produces its left-hand input, and an *outer relation* annotation indicates that the operator should execute at the site where its right-hand input is produced.

- *Select*[4] - The select operator applies a predicate to a relation and returns those tuples that satisfy the predicate. A select operator can be given either a *consumer* or *producer*

[2] For simplicity we use relational terminology when describing query operators. Analogous operations arise in object-oriented database languages that support associative queries such as OQL [Cat94].

[3] Other binary operators (such as set operations) can be annotated similarly to joins.

[4] Other unary operators (such as aggregations and projections) can be annotated similarly to selections.

annotation, indicating that it should be executed at the site of its parent or child operator respectively.

- *Scan* - The scan operator simply produces all of the tuples in a relation. A scan can be annotated in one of two ways: *primary copy* or *client*. A *primary copy* annotation indicates that the scan should be executed at the server where the relation resides. A *client* annotation indicates that the scan should be run at the site where the query is submitted, accessing data from the local cache if present; any missing data are faulted in from the server where the relation resides.[5]

At runtime, the logical annotations are bound to actual sites in the network. First the locations of the *display* and *scan* operators are resolved; then, the locations of the other operators are resolved given their annotations. Of course, it is possible for this binding process to result in operators with differing logical annotations being executed at the same physical site. It is, however, necessary to retain the distinct logical annotations in a query plan, as queries can be submitted at different sites, and relations can migrate to different servers over time. The implications of reusing annotated plans are discussed in Section 5.

2.2 Policy Definitions

The data-shipping, query-shipping, and hybrid-shipping policies can be defined by the limitations they place on assigning site annotations to the operator nodes of a query plan. This view of the three policies is shown in Table 1. For every operator used in this study, the annotations that are allowed by each policy are listed. The three policies are discussed briefly in the following subsections.

	data shipping	query shipping	hybrid shipping
display	client	client	client
join	consumer (i.e., client)	inner or outer relation	consumer, inner 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

2.2.1 Data Shipping

Data-shipping (DS) specifies that all operators of a query should be 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(a). The annotation of every operator is shown in italics, and the shading of the nodes indicates that every operator is executed at the client.

2.2.2 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 queries are completely evaluated at the server. There is, however, no recognized definition of query-shipping for systems with multiple servers. For this work, we define query-shipping (QS) as the policy that places *scan* operators on the servers at which the primary copies of relations reside, 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 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 1(b).

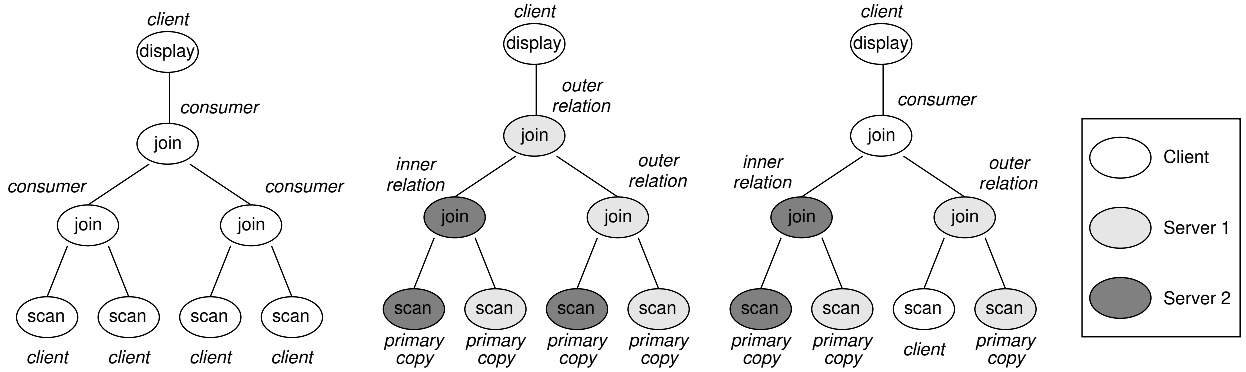
2.2.3 Hybrid Shipping

A hybrid-shipping (HY) approach allows each operator to be annotated in any way allowed by data-shipping or by query-shipping. Of the three policies therefore, hybrid-shipping has the most flexibility in producing plans. To guarantee that a binding of operators to sites can be determined at runtime, the optimizer must take precautions to generate only *well-formed* plans. A well-formed plan has no cycles, and as a consequence, there is a path (via 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 for example, if an operator A produces the input of an operator B, and the site annotation of A is *consumer* and of B is *producer*. Such a case could not be resolved by the simple scheme for binding operators to sites described above. Fortunately, because the query plans are trees, only cycles with two nodes can occur, and therefore, it is very easy to "sort out" ill-formed plans during query optimization. Figure 1(c) shows an example (well-formed) hybrid-shipping plan. It is interesting to note that, as shown in Figure 1(c), 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).

3 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 a randomized query optimizer and a detailed simulation environment. The simulation model captures the resources (CPU, memory, disk, and network) of a group of interconnected client and server machines. The query optimizer is based on randomized two-phase query optimization, which combines simulated annealing and iterative improvement, as proposed by Ioannidis and Kang [IK90]. Optimization can be aimed at minimizing either

[5] If horizontal partitioning is used, a *scan* operator must be defined for every fragment of the relation. Partitioning, however, is not taken into account in this study, so the unit of a scan is an entire relation. Furthermore, it is assumed that relations are not replicated at multiple servers.



(a)Data-Shipping (b) Query-Shipping (c) Hybrid-Shipping

Figure 1: Example Annotated Query Plans

the cost or the response-time predictions for a query plan. The search space explored by the optimizer includes the full range of shipping strategies; it can, however, be constrained to produce only data-shipping or query-shipping plans.

In the following, we describe the query optimizer and its cost model, the simulation environment, and a set of simple benchmark queries that are used to evaluate the tradeoffs among the different shipping policies. The results of the performance study are then presented in Sections 4 and 5.

3.1 Query Optimization

3.1.1 Implementation of the Optimizer

The query plans that are evaluated in this study are obtained using randomized two-phase query optimization (2PO) [IK90]. Randomized query optimization was chosen for the following reasons. First, randomized approaches have been shown to be successful at finding very good join orderings [IK90] and generating efficient plans even 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 approach optimizes very complex queries in a reasonable amount of time. It takes, for example, 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 practice as well), it is necessary only that the generated plans be "reasonable" rather than truly *optimal*. To minimize the impact of randomness on the results, all of the experiments reported here were executed repeatedly and confidence intervals for every data point were computed.

The optimizer first chooses a random plan from the desired search space (i.e., data, query, or hybrid-shipping) and then tries to improve the plan by iterative improvement (II) and simulated annealing (SA).[6] On each step, the optimizer performs one transformation of the plan. The possible moves are the following (where A, B, and C denote either temporary or base relations):

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* annotation of a *join* to *consumer*, *outer relation*, or *inner relation*.
6. Change the *site* annotation of a *select* from *consumer* to *producer* or vice versa.
7. Change the *site* annotation of a *scan* from *client* to *primary copy* or vice versa.

The optimizer can be configured to generate plans from one of the three policies 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 placed 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 placed 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.

3.1.2 Cost Model

In order to effectively optimize queries in a distributed environment it is necessary to have a reasonably accurate cost model. The cost model that we used is capable of estimating both the total cost and the response time of a query plan for a given system configuration. The *total-cost* estimates are based on the model of Mackert and Lohman [ML86]. The *response-time* estimates are generated using the model of [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 of the operators of a plan are executed sequentially, then the response time of a query is identical to its cost. However, if parallelism is exploited, then the response time of a query can be lower than the total cost. *Independent* parallelism can arise between

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

operators of different sub-trees of a plan; e.g., scan operators on different base relations can be executed in parallel. 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 way, a consumer can run in parallel with its producer operators.

3.2 Simulation Model

3.2.1 Execution Model

All experiments were carried out using a detailed simulation model. The simulator is written in C++ using the CSIM simulation toolkit. The simulator models a heterogeneous, peer-to-peer database system such as SHORE [C+94] and provides a detailed model of *query processing* costs in such a system. At present, it does not model concurrency control and transaction management functions that would be required to support non-query workloads. For this study, the simulator was configured to model a client-server system consisting of a single client and one or more servers. The impact of multiple clients in the system is modeled by placing additional load on the server resources and/or restricting the memory available for join processing. Clients and servers are similar in that they both have memory, CPU, disk resources, a buffer manager, and a query execution engine, but they differ in the following ways:

- Queries are submitted by an application at the client (even though some operators may execute at the server). The results of all queries are displayed at the client.

- The client's disk is used as a cache (i.e., to temporarily store copies of relations or relation parts that are brought in from the server), and for temporary storage for join processing. No primary copies of relations are stored at the client.

- The servers are responsible for managing primary copies of relations. These are stored on disk at the server. The primary copy of each relation resides on a single server (i.e., there is no declustering). Server disks are also used as temporary storage for join operators. In this study, there is no inter-query caching at servers. That is, when a server obtains data from another site, it uses it only for the duration of the current query.

Query execution is based on an iterator model, similar to that of Volcano [Gra93]. Each query operator has an *open-next-close* interface; *open* prepares the operator (e.g., allocation of main memory and structures, initialization of scans, etc.), *next* is called repeatedly to yield tuples, and *close* terminates the operator and does some final housekeeping (e.g., freeing memory). Each operator obtains data by calling the *next* functions of its children operators. For this reason, data flow is demand driven. When two connected operators are located on different sites, a pair of specialized network operators is inserted between them. These operators hide the details of shipping data across the network. Tuples are shipped across the network a page-at-a-time. In this case, pipelined parallelism can occur, because each producer has a process that tries to stay one page ahead of its consumer so that requests can be satisfied immediately.

As stated previously, clients support disk caching. If a disk is to be used both as a cache and for temporary storage, separate regions of the disk are allocated for each of these purposes. The disk cache is managed in large segments so that scans of cached relations can be done efficiently.

3.2.2 Resources and Model Parameters

Table 2 shows the main parameters that can be used to configure the simulator; the parameter settings used in this paper are described in Section 4.1. Every site has a CPU whose speed is specified by the *Mips* parameter, *NumDisks* disks, and a main-memory buffer pool. The CPU is modeled as a FIFO queue. Disks are modeled using a detailed characterization that was adapted from the ZetaSim model [Bro92]. The disk model includes an elevator disk scheduling policy, a controller cache, and read-ahead prefetching. There are many parameters to the disk model (not shown), including: rotational speed, seek factor, settle time, track and cylinder sizes, controller cache size, etc. For the purposes of this study, the important aspect of the disk model is that it captures the cost differences between sequential and random I/Os. In addition to the time spent waiting for and accessing the disk, a CPU overhead of *DiskInst* instructions is charged for every disk I/O request.

Parameter	Value	Description
Mips	50	CPU speed (10^6 instr/sec)
NumDisks	1	number of disks on a site
DiskInst	5000	instr. to read a page from disk
PageSize	4096	size of one data page (bytes)
NetBw	100	network bandwidth (Mbit/sec)
MsgInst	20000	instr. to send/receive a message
PerSizeMI	12000	instr. to send/receive 4096 bytes
Display	0	instr. to display a tuple
Compare	2	instr. to apply a predicate
HashInst	9	instr. to hash a tuple
MoveInst	1	instr. to copy 4 bytes
BufAlloc	min or max	buffer allocated to a join

Table 2: Simulator Parameters and Default Settings

The base relations, cached relation copies, and temporary results are organized in pages of *PageSize* bytes. Pages are the unit of disk I/O and data transfer between sites. The network is modeled simply as a FIFO queue with a specified bandwidth (*NetBw*); the details of a particular technology (i.e., Ethernet, ATM, etc.) are not modeled. The cost of a message involves the time-on-the-wire which is based on the size of the message, and both fixed and size-dependent CPU costs to send and receive which are computed from *MsgInst* and *PerSizeMI*.

In addition to the costs for system functions such as messages and I/Os, the costs associated with the query operators are also modeled; i.e., the CPU cost of displaying, comparing, hashing, and moving tuples in memory. All joins are processed using hybrid hashing [Sha86]. The amount of memory allocated to joins on a site is specified by *BufAlloc*[7].

3.3 Workload Specification

In order to examine the different execution strategies under a range of system configurations and settings, we use a benchmark consisting of 2-way and 10-way joins. The simple 2-way join query is used to explore fundamental tradeoffs between data-shipping and query-shipping. The more complex query is used to examine further effects of the basic policies and to provide sufficient latitude for the hybrid-shipping policy to generate plans that differ significantly from those generated by the pure policies.

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 in all temporary relations and in the query result is also 100 bytes. All joins are equi-joins. We have experimented with a variety of join graphs and join selectivities. For simplicity however, in this paper we focus on results for one particular type of query, as the effects described in Section 4 were seen in varying degrees, for all query types we investigated. Additional experiments can be found in [KF95].

The benchmark queries are *chain* joins with *moderate* selectivity. These queries can be thought of as "functional" joins, as would be used to reconstruct normalized data. 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; inter-operator parallelism can be exploited well in such queries. By moderate selectivity we mean that a join of two equal-sized base relations returns a result that is the size and cardinality of one base relation. This selectivity, besides being realistic for many queries, simplifies the analysis of the experimental results. Selectivity, however, plays a large role in the tradeoffs between the various approaches. In the experiments that follow results that would change significantly with different selectivities are noted.

4 Experiments and Results

In this section we present experiments that analyze the tradeoffs between data-shipping, query-shipping and hybrid-shipping. Many factors are varied in these experiments, including the complexity of the query, the server load, the number of servers, the amount of client caching, and the amount of memory allocated to joins. Before presenting the experiments and results, we first briefly describe the metrics used, as well as the simulation parameter settings and their implications.

4.1 Settings and Environment Details

The default simulation parameter settings used in this study are shown in Table 2. The settings are largely based on those used in previous studies; e.g., [Fra96, PCV94]. Several details deserve additional explanation. First, the total amount of memory allocated to a query at each site is equal to the allocation of memory for joins, plus some small overhead (e.g., for inter-operator communication buffers). The amount of memory allocated for joins depends on whether the joins of the query are to be executed with minimum or maximum allocation (i.e., the *BufAlloc* parameter). Minimum and maximum allocations are defined according to Shapiro [Sha86]. Maximum allocation allows the hash table for the inner relation to be built entirely in main memory. Minimum allocation reserves $\sqrt{F \times M}$ buffer frames for a join ($F = 1.2$ is a fudge factor, M is the size of the inner relation) and requires the inner and outer relations be split into partitions. All of these partitions (except for one) must be written and read to/from temporary storage.

In the experiments that follow it is assumed that all main-memory buffers are empty at the beginning of a query execution. Even when the memory of a site is large enough to provide the maximum allocation for joins, no data is cached in that site's main memory across queries. Data that is cached at the client is assumed to be initially resident on the client's local disk. As a result, disk I/O is always required to read the base relations from either a client or a server disk.

As stated in Section 3.2, the simulator contains a detailed disk model. In this study, we use the settings from [PCV94], which were intended to model a Fujitsu M2266 disk drive. The average performance of the disk model with these settings is roughly 3.5 msec per page for sequential I/O, and 11.8 msec per page for random I/O; these values were obtained by separate simulation runs to calibrate the cost model of the optimizer. To simulate additional server load and multiple clients, an extra process issuing random disk read requests is run at servers in some experiments. The request rate of this process can be varied to achieve different disk utilizations.

Two metrics are used when describing the results of the experiments: communication cost and response time. The first metric, expressed as the number pages sent, measures the average amount of data sent over the network during the execution of a query. This metric is useful for comparing the performance of the algorithms in a communication-bound environment such as the Internet. The second metric, response time, is used to measure the performance of the policies in a local-area, high-speed network (100 Mbit/sec). Response time is defined as the average time in seconds from the time that a query is initiated until the time its last result tuple is displayed at the client. For all experiments the query optimizer was configured to generate plans that minimized the metric being studied. Furthermore, as stated previously, the experiments were executed repeatedly so that the 90% confidence intervals for all results presented were within $\pm 5\%$.

4.2 2-Way Joins, Single Server

In the first set of experiments we focus on 2-way joins executed with a single client and a single server. This set of experiments establishes the main intuitions behind the tradeoffs of the different shipping policies. The tradeoffs are similar, but

[7] The allocation of memory in our experiments is described in Section 4.1.

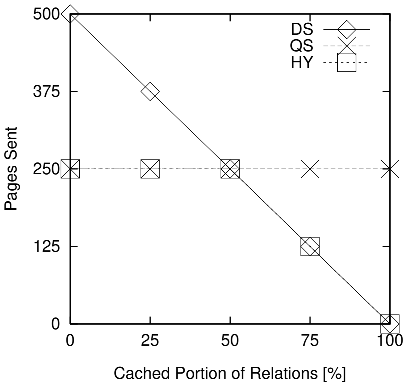


Figure 2: Pages Sent, 2-Way Join
1 Server, Vary Caching

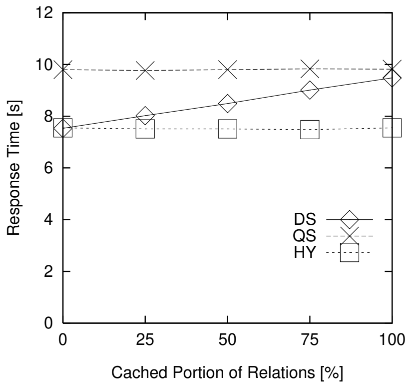


Figure 3: Resp. Time, 2-Way Join
1 S., Vary Caching, No Load, Min. Alloc.

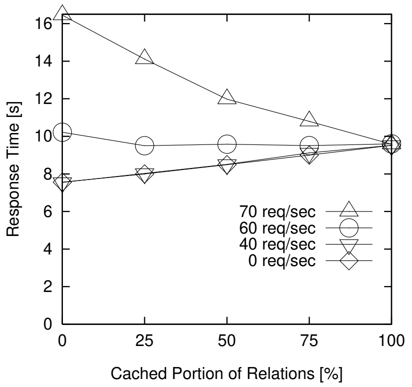


Figure 4: Resp. Time, DS, 2-Way Join
1 S., Vary Load & Caching, Min. Alloc.

somewhat more complicated for the 10-way join queries, which are discussed in Section 4.3.

4.2.1 Communication Cost

Figure 2 shows how the volume of data sent across the network varies for each of the three policies as the percentage of the relations cached at the client is increased.[8] The volume of data sent by query-shipping (QS), which performs all operations except for *display* at the server, is independent of the caching at the client. Therefore, in this experiment QS always processes the join at the server and sends the result (250 pages) to the client. In contrast, data-shipping (DS) and hybrid-shipping (HY) are both sensitive to the caching at the client. When no data is cached, DS must fault in both base relations from the server for join processing at the client, resulting in a communication volume of 500 pages, or twice as much as QS. The volume of data sent by DS decreases linearly as the amount of data cached at the client is increased, however. DS performs all scans at the client, faulting in only that data which is not cached locally. In this experiment, since the size of the join result is equal to the size of a base relation, the cross-over point between DS and QS is when half of each of the two base relations is cached at the client. The hybrid approach produces the same plans as QS before this point and as DS after, so it matches the best of the two pure policies in all cases here.

It should be emphasized that the specific cross-over point shown in Figure 2 results from the use of functional joins whose results are the same size as a base relation. This cross-over point would move to the right if the join result size was smaller than a base relation, and would move to the left if it was larger than a base relation. In any case, a hybrid approach would still be able to minimize its communication requirements to equal the lower of the two pure policies.

4.2.2 Response Time, Minimum Allocation

While the results for communication volume are fairly straightforward, the performance of the policies when communication is not a bottleneck is quite different. Figure 3 shows the response time for the two-way join query when the join is given the *minimum* memory allocation. Recall that when the join has only minimum memory allocation, it uses the disk to store partitions of the relations, thus placing additional, random load on the disk where the join is processed.

In this experiment, QS shows the worst performance because it always executes the scan and the joins at the server, and therefore, incurs high disk contention. Furthermore, the I/Os for join processing interfere with the sequential I/O patterns of the scan, resulting in a substantial degradation in I/O performance. In fact, the best plan in this situation is to perform the I/Os for the base relations at the server and to execute the join at the client, thereby exploiting disk parallelism and avoiding contention. When no data is cached at the client, DS, does exactly this. It executes the scan operators at the client, but since no data is cached, the I/O for the scans is performed at the servers. As caching is increased, however, DS reads more base data from the client's disk, and incurs more disk contention. As a result, the performance of DS steadily worsens until at 100% cached data, it performs only slightly better than QS (its slight advantage is because it does not need to send the result over the network to the client). Unlike DS, the HY approach is not forced to use cached data whenever it exists. As a result, HY finds the best plan in all cases here, regardless of the contents of the cache at the client; neither DS nor QS is able to do this.

The results of the previous experiment suggest that data caching on client disks is harmful when a pure DS policy is used. Clearly, this is not always the case. Figure 4 shows the performance of data-shipping plans when different loads are placed on the server's disk. These loads model the impact of contention for the server disk by multiple clients. The lowest line in Figure 4 is identical to the DS line in the previous figure — there is no external load placed on the server disk. When an external load of 40 requests/sec is placed on the server disk (i.e., 50% utilization), the performance of DS is similar to that when the server's disk is unloaded. When the load is increased further, however, the benefits of off-loading the server disk outweigh the disadvantages of the added contention on the client's disk, so the performance of DS can improve with the increase in client caching. In this case, with a server

[8] In this experiment, contiguous regions of relations are cached. For example, with a caching percentage of 25%, the first 25% of each relation is cached on the client's disk.

disk load of 60 requests/sec (76% utilization) DS performs slightly better with caching and when the load is increased to 70 requests/sec (90% utilization), the performance of DS improves significantly with increased caching. Although not shown in the figure, HY performs as before under light loads (it is better than both DS and QS) and tends towards the performance of DS under higher server loads. That is, as the load on the server is increased, HY moves more work to the client. QS, of course, does not have this flexibility, so its performance suffers dramatically as the server disk load is increased (i.e., the response time of QS is 19 seconds for 40 requests/sec and 36 seconds for 60 requests/sec).

4.2.3 Response Time, Maximum Allocation

The previous results showed the impact of exploiting client resources for joins and/or scans when only the minimum amount of memory is provided to the join. In this experiment, we examine the performance of the policies when joins are provided the *maximum* allocation, that is, when the hash table for the inner relation fits entirely in memory. In this case no disk resources are required for intermediate processing of joins so the placement of joins and scans on the same site does not result in contention or interference at the disk.

Figure 5 shows the response times of the three polices as client (disk) caching is increased, when the join is provided the maximum memory allocation. Similar to the communication costs described previously, QS performs better than DS when no data is cached, but its performance remains constant as the amount of cached data is increased, while the performance of DS improves linearly. One difference between this case and the communication-only case is that here, the cross-over point is slightly beyond 50% cached data. Even though QS and DS send the same volume of data over the network at this point, DS pays additional overhead because in our model, DS faults in base data a page at a time, while QS is able to overlap some communication and join processing. Constructing a DS approach to allow similar overlap is possible, but requires some sophisticated buffering techniques.

In this experiment, the optimizer generates a relatively poor plan for HY when 75% of the data is cached. This is because the cost model is not able to accurately predict the overlapping of communication and join processing. In this case it assumes that these costs can be fully overlapped, while in the simulator, such complete overlap is rarely attained.

The results are similar when the server disk is loaded (not shown), but the cross-over point for DS and QS is further to the left. For example, with an external server disk load of 60 requests/sec, the cross-over occurs when between 20% and 25% of the base relation data is cached at the client disk. With the maximum allocation, join processing and scan processing do not interfere at the disk, so DS and HY benefit from off-loading scan I/O from the server disk.

4.3 10-Way Joins, Multiple Servers

In this section we investigate the performance of the three execution policies when the complex, 10-way join queries are

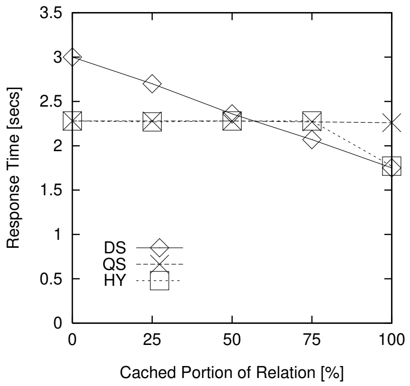


Figure 5: Resp. Time, 2-Way Join
1 Server, Vary Caching, No Load, Max. Alloc

processed in a system with multiple servers. Compared to the previous experiments, two new effects can be observed with the addition of servers and multiple join operators: 1) in some situations, costs for server-server communication are incurred; and 2) *independent* parallelism among operators in addition to *pipelined* parallelism can be exploited. In the following experiments, the ten base relations used in a query are placed randomly among the servers (ensuring that each server has at least one base relation). The data points presented below represent the average of many such random placements.

4.3.1 Communication Cost

As with the previous experiments, we begin by examining the communication costs of the three policies when the optimizer is configured to minimize such costs. Figure 6 shows the volume of data sent (in pages) by each of the policies as the *number of servers* is increased, with no client caching. DS always brings all ten base relations to the client for processing regardless of the number of servers, so its message volume remains constant at 2500 pages here. In contrast, the communication costs of QS are highly dependent on the number of servers. With only a single server, QS processes all of the joins using no communication, sending only the result (250 pages) over the network to the client. As the base relations are spread over more servers, however, QS must send those relations between servers in order to process joins. Thus, as servers are added, the communication costs of QS increase until with ten servers, its communication costs equal that of DS. As was seen in the previous experiments, HY has communication costs equal to the lesser of the two pure policies. In this case, it is the same as QS until ten servers are present, at which all three policies are equal.

The non-linearity in the increase of communication costs for QS and HY arises due to the fact that the placement of relations on servers is done independently of the ordering of joins dictated by the query. If two relations are co-located at a server but do not share a common join attribute, the optimizer will not join them locally as the result would be a Cartesian product (5 million pages). Thus, for example, in the case where the relations are spread across two servers, several entire base relations (or join results) must typically be

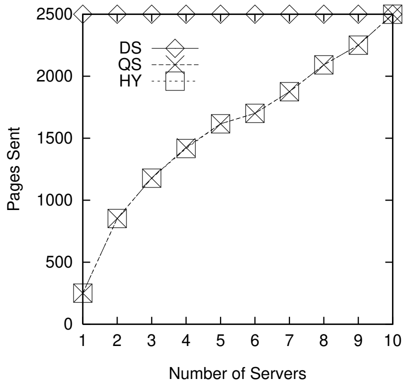


Figure 6: Pages Sent, 10-Way Join
Varying Servers, No Caching

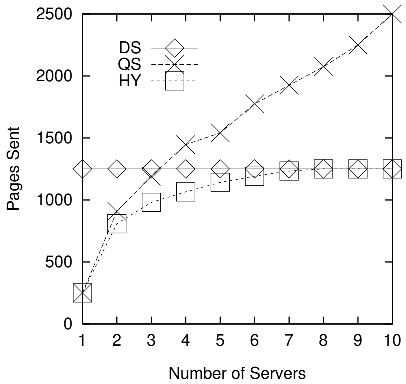


Figure 7: Pages Sent, 10-Way Join
Vary Servers, 5 Relations Cached

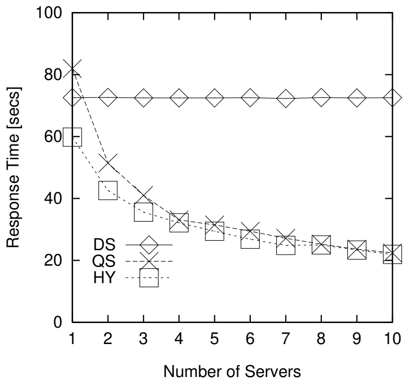


Figure 8: Resp. Time, 10-Way Join
Vary Servers, No Caching, Min. Alloc.

sent between the servers, resulting in a more than doubling of the communication cost incurred with a single server.

Turning to the effects of client caching, Figure 7 shows the communication costs when five of the ten relations are cached. In this case, the communication costs of DS are halved — it only needs to bring the remaining five relations across the network. In contrast, QS, which does not exploit the client cache, has the same costs as in the no-caching case. Therefore, beyond three servers, QS sends more data than DS here. Interestingly, in this case HY is actually able to send less data than either of the pure polices for many of the server populations. HY is able to save communication due to its flexibility to choose between base relations at servers or cached copies at the client. HY exploits this flexibility by placing joins at sites where they can be executed without communication. That is, if two relations that share a common join attribute are co-located on either a server or the client, HY can execute the join at that site, thus saving the cost of sending one of the relations to the other. This effect was not seen in any of the previous experiments due to the placement of all relations on a single server.

4.3.2 Response Time Experiments

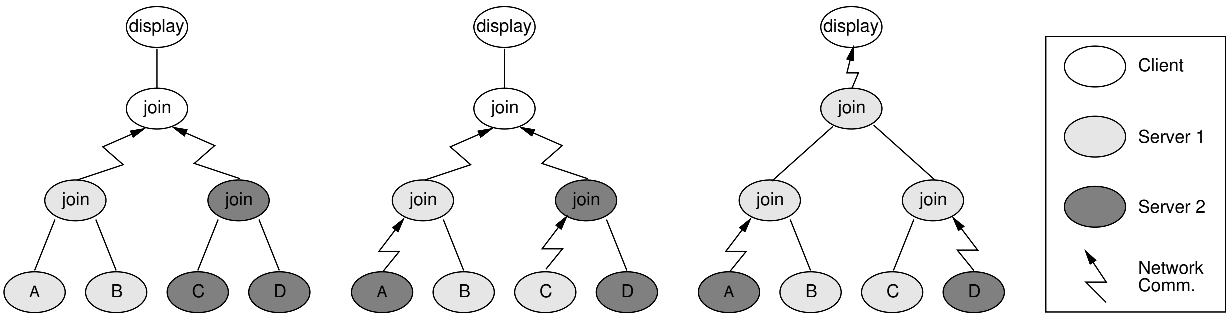
Figure 8 shows the response time results for 10-way joins when joins are given *minimum* memory allocation, as the number of servers is increased. There is no data cached at the client in this experiment. With minimum allocation, the cost of executing the hybrid-hash joins is the largest contributing factor to the response time. In this case, DS performs all join processing using the (single) client disk and all scan I/O using the disk(s) at the server(s). When there is only a single server, DS performs somewhat better than QS because (as seen in previous experiments) QS performs all join and scan I/O on a single disk, resulting in increased contention and interference. As additional servers are added to the system, however, the performance of QS improves greatly, because it is able to exploit parallelism among the server disks. In contrast, the performance of DS is largely independent of the number of servers, because the benefits of any parallelism associated with scanning at the servers is hidden by the cost of performing all joins at the client.

As can be seen in Figure 8, with small numbers of servers present in the system, HY performs better than both of the two pure policies because HY uses the client and the servers for query processing. In a system with one client and two servers, for example, HY executes 3 of the 9 joins of a query on each machine; DS, on the other hand, executes all 9 joins on the client, and QS executes 5 joins on one server and 4 joins on the other server. Of course, the benefit of using the client resources diminishes as more and more servers are added to the system. In this experiment, HY's advantage to QS is largely dissipated when there are more than three servers. It is important to note however, that these results are obtained assuming that the resources that each server machine has available to dedicate to processing this query is equal to the resources of the client. If the servers were more heavily loaded, the performance advantages of the hybrid approach would be greater and would extend over larger server populations.

The response times of the policies in the presence of client disk caching are not shown in this section because the effects have already been discussed in Section 4.2. That is, the performance of DS degrades the more data are cached because of increased contention on the client's disk; QS shows the same performance as in Figure 8 because it does not exploit client caching; and HY does not exploit client disk caching either because server resources are plentiful. Furthermore, the results when joins receive the *maximum* allocation are not shown. Because joins do not use disks in this case, the response time of a query is dominated by the file scans. If no data are cached at the client, therefore, DS is able to obtain the advantages of added servers (i.e., parallel execution of scans), and all three policies show roughly the same performance.

5 Optimization Issues for Flexible Systems

The previous sections have shown that a system should support a flexible query execution policy in order to fully exploit the benefits of client and server resources across a range of workloads and system configurations. As stated previously, increased flexibility increases the search space and, thus, the cost of query optimization. It is, therefore, desirable to precompile a query and avoid excessive computation prior to



(a) Static, compile time (b) Static, execution time (c) 2-Step, execution time

Figure 9: Example Static and 2-Step Plans for a 4-Way Join
Compile Time: A, B at Server 1; C, D at Server 2. Run Time: B, C at Server 1; A, D at Server 2.

every execution of the query. Pre-compilation, however, can produce sub-optimal plans because optimizations are carried out at *compile time* with imperfect knowledge of the *runtime* state of the system (caching, loads, etc.).

In this section, we study two types of optimization that pre-compile queries: static and 2-step. A *static* query optimizer generates a query plan based on the state of the system at compile time or based on general assumptions about the system state. *2-Step* optimization which has been proposed for distributed systems in [CL86, S+94] performs only some optimization decisions at compile time and then adjusts the partially optimized plan prior to query execution. We propose a 2-step optimizer that works as follows:

1. At compile time, generate an *incomplete* query plan including join orderings but no site annotations (e.g., using a randomized [IK90] or System-R-style [S+79] optimizer).
2. At execution time, carry out site selection and determine where to execute every operator of the plan (e.g., using simulated annealing [MLR90]).

2-Step optimization, therefore, takes into account that the system state can change rapidly; e.g., due to caching and varying loads. Like static optimization, however, it can produce a sub-optimal plan for a query because it carries out some optimizations (e.g., join ordering) based on imperfect knowledge of the runtime system state. In the following, we investigate the resulting performance when the runtime data location (e.g., caching, dynamic data migration and replication) cannot be predicted at query compilation time. Our focus here is on the quality of the plans so the overhead of dynamic site selection for 2-step optimization is not included in the performance results. Clearly, this overhead will ultimately play a large role for the overall performance of 2-step optimization, but it is highly dependent on the specific site selection algorithm used.

5.1 Communication Cost

We first examine the effects of data migration and caching on the communication costs of static and 2-step plans. The join ordering specifies the data flow during the evaluation of a query, and therefore, pre-compiling the join order with imperfect knowledge of data location can result in excessive communication costs. This limitation is illustrated by the example in Figure 9. The static plan shown in part (a) is a plan that minimizes communication under the assumption that relations A and B are co-located at Server 1 and that relations C and D are co-located at Server 2. In this case, the required communication is the sending of two join results to the client (to be joined there). If, however, the data were to migrate so that at runtime, relations B and C were co-located and relations A and D were co-located, then as shown in Figure 9(b), the statically generated plan would incur the additional cost of sending two relations to be joined. Assuming that all relations are joinable and that join results and base relations are the same size, the static plan performs twice as much communication at runtime than an optimal plan that performs $B \bowtie C$ at Server 1 and $A \bowtie D$ at Server 2; this optimal plan incurs only the cost of sending two temporary join results to the client.

The 2-step plan, shown in part (c), would use the same join ordering as the static plan, but it can be more flexible about where it places the join operators. In this example, this flexibility results in a reduction of the communication penalty. Nevertheless, by being constrained to use the pre-compiled join ordering, the 2-step plan still performs 50% more communication than an optimal plan.

The above example demonstrates that 2-step optimization has the potential to generate plans with excess communication in the presence of data migration among servers. It should be noted, however, that 2-step optimization can be effective in reducing communication for one important aspect of flexible client-server database systems; namely if at runtime copies of data are cached at the client that submits a query, 2-step optimization has the flexibility to exploit the cached data to reduce communication. This aspect of 2-step optimization is promising, because data caching is likely to be much more dynamic than data migration.

5.2 Response Time Experiments

The previous section addressed the communication costs of static and 2-step optimization. In this section, we examine the response time of compiled plans when the number of

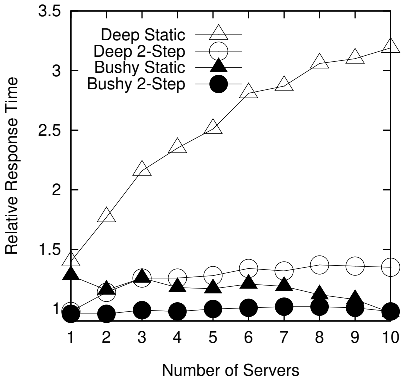


Figure 10: Relative Response Time, 10-Way Join
Vary Servers, No Caching, Min. Alloc, Deep and Bushy Plans

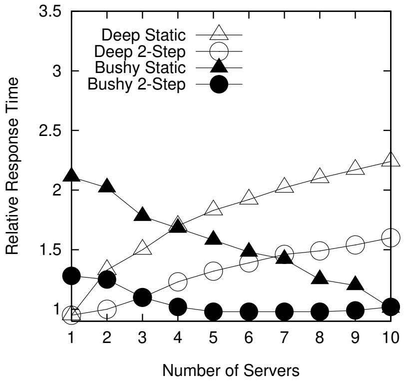


Figure 11: Relative Response Time, *HiSel* Query
Vary Servers, No Caching, Min. Alloc, Deep and Bushy Plans

servers storing base relations is unknown at compile time. As shown in Section 4.3, large benefits can be gained by exploiting multiple servers. The effective use of multiple servers, however, strongly depends on the *shape* of the join graph. A *bushy* plan can exploit multiple servers by allowing joins to run in parallel, but it does so at the potential cost of increasing the sizes of intermediate relations. In contrast, a *left-deep* plan minimizes the intermediate relations but cannot exploit parallel resources.

Figure 10 shows the response times (relative to an *ideal* plan) of left-deep and bushy plans for the 10-way join query when the *minimum* memory allocation is given to each join. To obtain static and 2-step left-deep plans, the optimizer was told at compile time that the database was *centralized* on a single site. The bushy plans were obtained by telling the optimizer that the database was *fully-distributed*— i.e., that each relation was stored on a separate server. In this experiment the static left-deep plans pay a huge penalty because due to the centralized assumption, all joins are executed on a single site. When 2-step optimization is applied to these plans, this penalty can be mitigated because the runtime site selection can redistribute the joins. However, the deep 2-step plans still perform worse than the bushy plans with multiple servers, because they cannot exploit independent parallelism among the joins. A static bushy plan suffers when few servers are present, because it fails to exploit client resources, and also with more servers because it does not evenly distribute the load across the available servers. In contrast, the 2-step bushy plan is able to perform nearly as well as an *ideal* plan for all server populations here.

As stated in Sections 3.3 and 4.2, the join selectivity can have a substantial impact on the performance of a query processing strategy. The weakness of bushy plans become apparent if the join selectivity is high. As can be seen in Figure 11, with small number of servers, the bushy plans perform poorly for a *HiSel* 10-way join in which only 20% of the tuples of every input relation paritcipate in the output of a join. As servers are added, however, a bushy 2-step plan performs well for this query, too, because the extra work that it does is split across many servers and is largely done in parallel.

6 Related Work

As stated in the introduction, both data-shipping and query-shipping are commonly used in distributed database systems. Data-shipping is used in most object database systems, and it has been investigated, e.g., in [D$^+$90, Fra96]. Query-shipping, on the other hand, is used in most traditional relational systems. ADMS [R$^+$95] is an example of a system that uses an extended query-shipping architecture: query results are cached at clients, and a query can be answered at the client if it *matches* the cached results of a previous query; if it does not match, the query is executed at the servers.

Current trends towards very large systems with many sites have motivated the design of new flexible architectures that, like hybrid-shipping, allow query processing on clients and servers; examples are Orion [J$^+$90] and Mariposa [S$^+$94]. Furthermore, SHORE [C$^+$94] provides a mechanism to allow the inclusion of a flexible query execution policy.

There have been several closely related studies of query processing in local-area and client-server database systems. Carey and Lu studied join algorithms in local networks [LC85] and devised methods to load balance a system if data is replicated at several sites [CL86]. Among the first to study query processing specifically in a client-server environment were Hagmann and Ferrari [HF86]. They investigated different ways to split the functionality of a DBMS (e.g., query parsing, optimization, and execution) between client and server machines.

Related work on multi-node query processing includes techniques devised for parallel databases. In particular, cost models and efficient query optimization techniques devised for parallel database systems (e.g., [SE93, GW93]) can be adapted to distributed database systems.

7 Conclusion

In order to build high-performance database systems that combine the best aspects of the relational and object-oriented approaches, it is necessary to design flexible client-server architectures that can fully exploit the client and server resources available in the system. As an initial step towards this goal, we have investigated the tradeoffs between three paradigms for query execution in client-server database systems: *data-shipping* which is commonly used in ODBMS; *query-shipping* which is used in traditional RDBMS; and *hybrid-shipping* which combines the features of data and query shipping. We first showed that these three policies can be characterized by the way they restrict the assignment of site annotations to query operators during query optimization. We then evaluated the performance of the three policies in a series of experiments using a randomized query optimizer and a detailed simulation model.

A fundamental difference between data, query, and hybrid-shipping is the degree to which they exploit client resources. Client resources can be used in two ways: 1) query operators can be executed at clients, and 2) data can be cached at clients (in memory and on disk). Data-shipping makes the most possible use of client resources; it executes all operators at clients and uses client-cached data whenever it is present. Query-shipping, on the other hand, neglects client resources; it performs all work at servers. Hybrid-shipping allows the flexible use of client resources by executing query operators on clients and/or on servers; it at least matches the best performance of data and query shipping and outperforms both in many situations.

The performance experiments showed that the advantages of hybrid-shipping were due to its flexible use of client resources in both ways, namely for executing query operators and using cached data. Hybrid-shipping executes query operators at clients if client resources are at least at parity with server resources, but it can execute operators on servers when parallel and/or plentiful server resources are available. Hybrid-shipping was also shown to exploit caching to reduce communication costs in some cases, but it does not use cached copies of data if the relations used in a query are co-located at a server. Likewise, hybrid-shipping can exploit caching to reduce interaction with heavily loaded servers, but it ignores cached copies of data if server resources are plentiful and the use of cached data at the client would increase the response time of a query by causing contention on the client's disk.

While the performance experiments showed the potential advantages of a hybrid-shipping policy, the implementation of such a policy raises a number of difficult issues. In particular, hybrid-shipping results in more complex query optimization, and pre-compiled query plans can be sensitive to changes in the system state and data location. To address these issues, we investigated the use of 2-step query optimization. Our initial experiments indicate that a 2-step optimization approach is promising. Dynamic site selection enables a 2-step optimizer to make use of client-side caching and client resources whenever this is beneficial. We found significant inefficiencies only for network-bound query processing; in these situations, however, it might well be affordable to carry out more complex on-the-fly optimizations since these optimizations only require additional client CPU cycles.

As an initial study in the area of merging relational and object-oriented database techniques at the *architectural* level, this paper focused on a simple workload consisting only of bulk queries. In future work, we intend to analyze the effects of navigation-based access and updates. In addition, we plan to study the impact of caching and the use of the aggregate main memory of the system in multi-query workloads. We are currently constructing a prototype query execution engine on top of the SHORE storage system to investigate these issues.

References

[Bro92] K. Brown. PRPL: A database workload specification language. Master's thesis, Univ. of Wisconsin, 1992.

[C+94] M. Carey, et al. Shoring up persistent applications. In *ACM SIGMOD Conf.*, Minneapolis, 1994.

[Cat94] R. G. G. Cattell, editor. *Object Database Standard*. Morgan-Kaufmann Publishers, San Mateo, 1994.

[CL86] M. Carey and H. Lu. Load balancing in a locally distributed database system. *ACM SIGMOD Conf.*, Washington, 1986.

[D+90] D. DeWitt, et al. A study of three alternative workstation server architectures for object-oriented database systems. *16th VLDB Conf.*, Brisbane, 1990.

[Fra96] M. Franklin. *Client Data Caching*. Kluwer Academic Press, Boston, 1996.

[GHK92] S. Ganguly, W. Hasan, and R. Krishnamurthy. Query optimization for parallel execution. *ACM SIGMOD Conf.*, San Diego, 1992.

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

[GW93] S. Ganguly and W. Wang. Optimizing queries for coarse grain parallelism. Technical report, Rutgers University, 1993.

[HF86] R. Hagmann and D. Ferrari. Performance analysis of several back-end database architectures. *ACM TODS*, 11(1), 1986.

[IK90] Y. Ioannidis and Y. Kang. Randomized algorithms for optimizing large join queries. *ACM SIGMOD Conf.*, Atlantic City, 1990.

[J+90] B. Jenq, et al. Query processing in distributed ORION. *2nd EDBT Conf.*, Venice, 1990.

[KF95] D. Kossmann and M. Franklin. A study of query execution strategies for client-server database systems. Technical Report, University of Maryland, 1995.

[Kul94] K. Kulkarni. Object-oriented extensions in SQL3: A status report. *ACM SIGMOD Conf.*, Minneapolis, 1994.

[LC85] H. Lu and M. Carey. Some experimental results on distributed join algorithms in a local network. *11th VLDB Conf.*, Stockholm, 1985.

[LVZ93] R. Lanzelotte, P. Valduriez, and M. Zait. On the effectiveness of optimization search strategies for parallel execution spaces. *19th VLDB Conf.*, Dublin, 1993.

[ML86] L. Mackert and G. Lohman. R* optimizer validation and performance evaluation for distributed queries. *12th VLDB Conf.*, Kyoto, 1986.

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

[PCV94] J. Patel, M. Carey, and M. Vernon. Accurate modeling of the hybrid hash join algorithm. *ACM SIGMETRICS Conf.*, Nashville, 1994.

[R+95] N. Roussopoulos, et al. The ADMS project: Views "R" us. *IEEE Data Engeneering Bulletin*, 18(2), 1995.

[S+79] P. Selinger, et al. Access path selection in a relational database management system. *ACM SIGMOD Conf.*, Boston, 1979.

[S+90] M. Stonebraker, et al. Third generation data base system manifesto. Technical Report, UC Berkeley, 1990.

[S+94] M. Stonebraker, et al. Mariposa: A new architecture for distributed data. *IEEE Conf. on Data Engineering*, Houston, 1994.

[SE93] J. Srivastava and G. Elsesser. Optimizing multi-join queries in parallel relational databases. *2nd PDIS Conf.*, San Diego, 1993.

[Sha86] L. Shapiro. Join processing in database systems with large main memories. *ACM TODS*, 11(9), 1986.