Cache Investment Strategies*

Michael J. Franklin
University of Maryland
franklin@cs.umd.edu

Donald Kossmann
University of Passau
kossmann@db.fmi.uni-passau.de

Univ. of MD Technical Report CS-TR-3803 and UMIACS-TR-97-50

May 1997

Abstract

Emerging client-server and peer-to-peer distributed information systems employ data caching to improve performance and reduce the need for remote access to data. In distributed database systems, caching is a by-product of query operator placement — data that are brought to a site by a query operator can be retained at that site for future use. Operator placement, however, must take the location of cached data into account in order to avoid excessive data movement. Thus, there exists a fundamental circular dependency between caching and query optimization. In this paper, we identify this circularity and show that in order to break it, query optimization must be extended to look beyond the performance of a single query. To do so, we propose the notion of Cache Investment, in which a sub-optimal plan may be generated for a particular query in order to effect a data placement that is beneficial for subsequent queries. We develop a framework for integrating Cache Investment decisions into a distributed database system without changing basic components such as the query optimizer's search strategy, the query engine, or the buffer manager. We then describe several cache investment policies, and analyze them using a detailed simulation model. Our results show that cache investment can significantly improve the overall performance of a system compared to the static operator placement strategies that are used by today's database systems.

Index terms - Distributed Databases, Client-Server Databases, Query Processing, Query Optimization, Caching, Database System Performance.

1 Introduction

Caching has emerged as a fundamental technique for ensuring high performance in distributed database systems. It is particularly important in large systems with many clients and servers because it reduces communication costs and off-loads shared server machines. Caching has been successfully integrated into many commercial and research database systems, data warehouses, WWW browsers, and Internet proxy-servers. Despite this widespread acceptance, many aspects of the integration of caching with query processing are not yet well understood. This paper focuses on one key aspect of this integration, namely, the circular dependency that exists between caching and query optimization — more specifically, between caching and query operator site selection.

In [FJK96], we introduced a query execution model called *hybrid-shipping*, which is able to exploit the presence of cached data through the use of flexible operator site selection. The goal of operator site selection

*This work was partially supported by NSF Grant IRI-94-09575, an IBM SUR award, and a grant from Bellcore. Donald Kossmann was supported in part by the Humboldt-Stiftung, UMIACS, and DFG Grant Ke 401/7-1.

is to place query operators (e.g., scans, selects, and joins) among the nodes of a distributed system in a way that minimizes the cost of a given query for a given data placement. Caching impacts site selection because it dynamically changes the data placement, and hence the performance of certain distributed query plans. Caching, however, is also impacted by site selection. If a query plan causes data to be brought to a site for processing, then those data can subsequently be cached at that site. Depending on site selection, therefore, the contents of a cache can change dramatically as the result of the execution of a query. The influence of this circular dependency between caching and site selection on the performance of a system is demonstrated by the following two examples:

Example 1 A request to compute a join between relations A and B is submitted at a client workstation. Both relations have 10,000 tuples of size 100 bytes each (1 MB), and the result of the join is estimated to have 9,000 tuples of 100 bytes (0.9 MB). No tuples of either relation are initially cached at the client. Relation A is stored at Server *I*, and relation B is stored at Server *II*; the relations are not partitioned, and no copies of data from the relations are available on any other sites. The three machines are connected by a slow, wide-area network.

One possible plan is to ship a copy of relation A to Server *II*, to compute the join there, and to ship the result to the client. This plan has communication costs of 1.9 MB. An alternative is to execute the join at the client, and ship copies of relations A and B from the servers to the client. This plan has slightly higher communication costs of 2 MB. In isolation, it would appear that the first plan is slightly preferable to the second. If, however, a subsequent query to join relations A and B with the same selectivity is posed at the client, the evaluation of this query would again require communication costs of at least 1.9 MB. In contrast, this second query could be performed with *zero communication costs* had the "sub-optimal" plan for the first query been chosen instead, as that plan would have enabled A and B to be cached at the client.

Example 2 A query is submitted at a client that selects, with a high-selectivity predicate, a few tuples from a very large relation C which is stored on Server *I*. If no copy of relation C is cached at the client, the best plan is to carry out the selection at Server *I* and to ship the few tuples that qualify to the client. An alternative is to ship relation C to the client and carry out the selection at the client. This plan has very high communication requirements, and the additional cost to ship the whole relation to the client will only pay off if relation C is used in many subsequent queries. Another problem with this alternative plan is that relation C might flood the client's cache and replace hot data (e.g., relations A and B) that are more likely to be used in subsequent queries.

The first example demonstrates that in some cases the optimizer should be forced to generate a sub-optimal query plan and *invest* resources to initiate the caching of data. Such an investment, while hurting the performance of a single query may enable the efficient execution of future queries. In contrast, the second example shows that in other cases, cache investment may dramatically hurt the response time of the current as well as future queries. These two examples demonstrate some of the potential advantages and pitfalls of cache investment. In this paper we analyze these tradeoffs in further detail. We present and evaluate several alternative cache investment policies. These policies take effect during query optimization and can easily be integrated into a system without changing basic components such as the optimizer's search strategy, the query engine, or the buffer manager.

Because caching establishes copies of data at a site, it can be seen as a form of replication. Replication has been thoroughly investigated in previous work; e.g., in [WJ92b, WJ92a, AZ93, Bes95, SAB+96]. Such algorithms however, cannot be directly applied to support caching — they are based on the *global* popularity of data in order to load balance the entire system and move data closer to a group of sites that frequently use the data. Caching, on the other hand, is specifically designed to support the query workload of a single site (e.g., a client workstation), and it must adapt quickly to instantaneous changes of the site's workload. Recently, dynamic, cost-based algorithms have been proposed for cache admittance and replacement in environments such as networks of workstations [SW97] and data warehouses [SSV96]. These latter approaches use cost and benefit estimates to determine the value of retaining cached copies of data, but they do so by moving data directly, independent of query optimization and processing.

Cache Investment, unlike previous caching and replication approaches, is based on the realization that caching decisions and query optimization are inherently related. A novel feature of our approach, therefore, is that caching decisions are affected indirectly, by influencing the query optimizer. In this way, caching decisions are better integrated with the cost estimations made by the optimizer and can exploit the optimizer's ability to examine and choose among a vast number of potential plans. Furthermore, our solution exploits the query processing engine to move data, obviating the need for a separate mechanism for effecting data placement.

The remainder of this paper is organized as follows. Section 2 describes the basic assumptions and the overall architecture for query processing and data caching used in this work. Section 3 defines several cache investment policies and shows how they can be integrated into a system. The policies were evaluated in an extensive simulation study. Section 4 describes the experimental environment, and Section 5 presents the results of the tradeoff analysis. Section 6 discusses related work in more detail. Section 7 presents conclusions.

2 Architecture and Assumptions

Our work is based on a client-server caching architecture in which queries are submitted, data is cached, and results are displayed at client workstations, while the primary copies of data reside on server machines. The techniques we present, however, can naturally be applied to other distributed database architectures, such as a symmetric peer-to-peer system like SHORE [CDF+94], in which every site acts as a client and/or as a server. We assume the use of a *hybrid-shipping* query execution model, which as shown in our earlier work, allows query processing to best exploit the resources of such a system [FJK96]. Hybrid-shipping is a flexible policy in which query processing can be performed at clients, servers, or various combinations of them according to the query plan produced by the optimizer. In the following, we describe the architecture of a hybrid-shipping system, focusing on the features that are relevant to cache investment.

2.1 Query Processing

As described in [FJK96], two key aspects of hybrid-shipping query processing are flexible site selection for query operators and the binding of such site selections at query execution time. With hybrid-shipping, queries are executed in an architecture that allows query operators to run on clients and/or on servers. This flexibility is in contrast to traditional data-shipping and query-shipping systems, which restrict query processing to occur solely at clients or servers respectively. The importance of operator placement flexibility was demonstrated in the two examples of the introduction: in Example 1, the operators of the queries should be executed at the client whereas the query of Example 2 should be executed at the server. Furthermore, as shown in [FJK96], there are cases where the operators of a single query should be split among clients and servers. At present, most client-server database systems do not provide the flexibility to choose among these options. A flexible approach, however, has been used in several recent experimental systems such as ORION-2 [JWKL90] and Mariposa [SAL+96], and is being integrated into an extended version of the SHORE storage manager [CDF+94] as part of the DIMSUM [1] project.

The second important feature of hybrid-shipping, the binding of operators to sites at query execution time, requires that the decision of where each operator of a query is to be executed (i.e., at a client or a server) be made when a query is prepared for execution. These decisions are made given knowledge of the contents of the client's cache and, if possible, of the load situation of servers. Obviously, run-time site selection is vital for making use of the client's cache; for example, to carry out a join at the client if copies of both relations are already cached. Run-time site selection is also needed to allow load balancing [CL86]. For interactive, ad-hoc queries, query optimization and site selection are both carried out at execution time. For pre-compiled queries that are part of, say, an application program, a two-step approach can be used, in which most optimization decisions (e.g., join ordering) are made at compile time, but site selection is carried out at execution time. Similar approaches have been proposed in [CL86, SAL+96, FJK96].

2.2 Cache Management

We study an architecture in which data can be cached in a client's main memory or on a client's local disk [FCL93]. We focus on the case where, as in many data-shipping systems, cached data consist of pages of base relations. Such physical caching is in contrast to the logical caching of data such as query or sub-query result caching (e.g., [RK86, CR94, SJGP90, KB94, DFJ+96]). More specifically, we assume that the database is partitioned into fragments and that individual pages of a fragment can be cached using a page-server architecture [DFMV90]. A fragment refers to any collection of pages that is stored permanently in a single file at a server; e.g., a relation or horizontal partition of a relation. Caching at a client is initiated by placing a scan operator of a query on the client. A scan takes a fragment as input and delivers a stream of tuples as output. If a scan is executed at a client, the pages of the fragment that are cached at the client

[1] See `http://www.cs.umd.edu/projects/dimsum` for more information.

are used — all other pages are *faulted in* from server(s) and can subsequently be cached at the client. In contrast, if the scan is executed at a server, the cache of the client is not used and no new pages of the fragment can be cached at the client. Caching cannot be initiated by any other operator (e.g., a join) as the input of all other operators is a sub-query result and therefore, is discarded after the execution of the query.

In this paper, we focus on the performance of cache investment when used in conjunction with an invalidation-based cache consistency policy. Under such a policy, the new version of a page is shipped to a client only upon request after the page has been updated, rather than propagating the new version of the page automatically to all clients that cache the page. Invalidation has been shown to be more robust than propagation across a wide range of workload and system scenarios [FCL97]. Callback locking is a prominent example of an invalidation-based policy, and it is used in several client-server database systems; e.g., ObjectStore [LLOW91] and SHORE [CDF+94].

3 Policies for Cache Investment

In this section we present policies for determining when and for which fragments the investment required to initiate caching should be made. These policies are invoked for each query that is submitted at a client, and can influence the way that operator site selection is done for that query. We describe two types of policies: (1) *static* policies, which correspond to query processing approaches that are used in current distributed database systems; and (2) *history-based* policies, which explicitly take into account the circular dependency between caching and query optimization that we have identified in the preceeding sections. Before describing these policies, however, we outline a general framework for making cache investment decisions.

3.1 Identifying Candidates for Caching

All cache investment policies require a mechanism for determining which fragments should be cached at the client. We refer to such fragments as *candidates*. When one or more candidate fragments are used in a query the policy *fires* in order to coerce the optimizer into generating a (potentially sub-optimal) plan that will result in the caching of those fragments at the client. Note that if a query uses no candidate fragments, or if all candidate fragments used by a query are already fully cached at a client, then the policy does not fire and query optimization is carried out as normal. The interaction of these policies with the query optimizer is described at the end of this section.

The decision of whether or not a fragment should be considered a candidate is a tradeoff between the cost of initiating the caching of that fragment (i.e., the *investment*) and the expected gain to be realized by caching the fragment (i.e., the ROI, or *return on investment*). The investment cost for caching a fragment is paid by a single query (i.e., the one that initiates the caching) and can be defined as the difference in response time or cost of the plan that results in the caching of the fragment and that of an optimal plan for the query. Thus, if the fragment is already cached, the investment is 0. In contrast, the ROI depends on *future* queries and *future* updates to the fragment. ROI can be defined as the cumulative savings in the response time (or communications costs) of future queries that can be achieved while the fragment remains cached. Thus, if the fragment is not used in a future query, the ROI is 0.

The policies we study differ in how they compute investment and ROI when choosing candidate fragments. Intuitively, an ideal Cache Investment policy would chose candidates based on perfect knowledge of both the investment and ROI for all fragments in the database. Under such an ideal policy, a fragment would be considered a candidate at a client if it met the following two criteria:

1. The ROI of the fragment is higher than the investment for the fragment.
2. The ROI of the fragment minus the investment for the fragment is higher than the ROI of the currently cached fragment(s) it would replace if it were brought in.

The first criterion ensures that investing in the fragment would produce a net gain. The second criterion ensures that only the most valuable fragments are kept in the client's cache. Of course, there are several problems with implementing such an ideal policy. Of primary importance is the fact that the ROI and investment costs of fragments can not be accurately known. The computation of investment depends on the cost model and estimates of the query optimizer, which are likely to be inaccurate. Calculating ROI is even more difficult, as it also depends on predictions of future behavior. All the policies studied in this work either estimate or assume a fixed value for ROI and investment of fragments. We refer to policies that use statistics from past queries to make investment decisions as *history-based* policies while those that use only fixed values are called *static* policies. In most situations, a history-based policy can adapt better to a client's workload than a static policy, but using the past is not always a good way to predict the future. Furthermore, generating and maintaining statistics requires additional computational overhead to carry out a history-based policy. In this paper, we study two static and two history-based policies; these four policies are described below.

3.2 Static Policies

Static policies assign fixed values for investment and ROI to fragments independent of any history. As stated above, such policies are typical of the way that existing systems are built: that is, without considering the long-term effects of interactions between caching and query optimization. In this work, we examine two static policies: the *Conservative* policy assigns values such that it never fires, and the *Optimistic* policy assigns values such that it always fires. These two static policies correspond to the traditional notions of query shipping and data shipping, respectively. Of course, a static policy is not able to adapt on-the-fly to a particular workload. Examining the limitaions of such policies shows where the benefits of a long-term view of cache investment lie. Thus, we use these policies primarily as baselines against which to compare the history-based policies defined in the next section.

Conservative Policy The Conservative policy assigns the ROI of every fragment to be 0 and the investment to be ∞ so it never considers any fragments to be candidates. Thus, the Conservative policy never fires, and query optimization is carried out in the conventional way. The optimizer places all scans at servers because placing scans at a client to initiate caching usually comes at an additional cost; as a result, the cache of a client is always empty. The behavior of the Conservative policy corresponds to that of traditional relational (i.e., query-shipping) database systems which do not employ caching.

Optimistic Policy The Optimistic policy is so named because it sets the ROI of all fragments to be ∞, and the investment to be 0. It therefore considers all fragments to be candidates and attempts to bring all fragments accessed by the query into the client's cache if they are not already there. The behavior of the Optimistic policy corresponds to that of a *data shipping* architecture which places all scans at the client in order to exploit client caching.

3.3 History-based Policies

In this section, we describe two simple history-based policies: *Reference-Counting* and *Profitable*. Both policies try to adapt to the workload at each client based on the past history of queries at that client. They differ in that the Profitable policy attempts to directly estimate the investment and ROI for fragments, while the Reference-Counting policy is simpler; it ranks fragments by their frequency of use, without explicitly calculating expected ROI's, and ignores investment costs.

3.3.1 Maintaining History Information

Both Reference-Counting and Profitable maintain history information about fragments at all client sites. The information stored for a fragment is a number that represents a *value* for the fragment based on the history of queries at that client. The way that values are assigned differs according to the particular policy being used. For both policies, the values of fragments at a client are adjusted after the execution of each query at that client. This adjustment is performed using *periodic aging by division*, as proposed in [EH84]. The value of every fragment is initially set to 0. As described in Equation 1, the value $V_t^i(j)$ of a fragment j at client i after the execution of query t is multiplied by an aging factor α ($0 \leq \alpha \leq 1$), and increased by a component $C_t(j)$ ($0 \leq C_t(j) \leq \infty$).

$$V_t^i(j) = C_t(j) + \alpha * V_{t-1}^i(j) \tag{1}$$

For both history-based policies, $C_t(j)$ is set to 0 if fragment j was *not* used in query t, and is set to a value ≥ 0 otherwise. α is a tuning parameter and determines the weight given to past queries: for $\alpha = 1$, all queries are given the same weight, if $\alpha < 1$, then recent queries are given more weight than past queries. In the extreme case ($\alpha = 0$), the value of a fragment is based entirely on the most recent query. As a result, with a smaller α a policy can adjust to changes in the workload more quickly, but it becomes more sensitive to transient changes in the workload at a client. Note that to reduce computational overhead, the re-computation of fragment values can be restricted to only those fragments whose value is above a certain threshold. When the value of a fragment drops below this threshold, the value is set to 0 and the fragment is ignored until it is again used in a query.[2]

[2] In the study that follows, we use a threshold of 0.01 for this purpose.

The value of a fragment is also adjusted to account for invalidations of cached pages due to updates. Recall that when using an invalidation-based cache consistency scheme, page copies are removed from client caches when the page is updated elsewhere. These invalidations selectively remove parts of a fragment from a client's cache, reducing the amount of time that the fragment remains cache-resident after an investment has been made to cache it. If a fragment is susceptible to being updated elsewhere, then it may not be a good fragment on which to invest. Thus, in order to factor updates into the calculation of fragment value, the server provides information on updates to the clients and the value of a fragment is reduced proportionally to the number of its pages that have been updated since the last time the fragment was used. In the extreme case, if all of the pages of a fragment have been updated, the value of the fragment is set to 0.

Given the description above, two questions must be answered to instantiate a history-based policy:

1. How is $C_t(j)$ computed?

2. When is a fragment considered to be a candidate?

We now describe the Reference-Counting and Profitable policies, focusing on the way that they address these two questions.

3.3.2 The Reference-Counting Policy

Reference-Counting is an extension of reference-based replacement policies used in database buffer management [EH84]. For Reference-Counting, the component $C_t(j)$ of equation 1 is set to 1 if any part of fragment j is used in query t (that is, if during the execution of query t at least one page of fragment j was accessed) and is set to 0 otherwise. Thus, the value of a fragment for Reference-Counting is a count of the number of queries in which a fragment is used, possibly weighted by the recency of those accesses as determined by the α parameter.

Once the Reference-Counting policy has computed the values of all fragments, it decides which ones to consider as candidate fragments. Unlike the ideal policy described in Section 3.1, Reference-Counting does not compute estimated ROI's for the fragments and it ignores the cost of investment. Instead, it decides on which fragments should be candidates based on the value it maintains for each fragment, the sizes of the fragments, and the size of the client's cache.

fragment	*value*	*size in pages*	*value/size*
A	200	50	4
B	600	200	3
C	200	100	2
D	100	100	1

Table 1: Example of Cache Value Computation

The Reference-Counting policy tries to maximize the value of the fragments stored in a client's cache using an approach similar to that of Bubba [CABK88] and [SJGP90], in which the *value/size* ratio is taken into account. A fragment is considered to be a candidate only if its value is greater than 0 and it would be fully or partially kept in a cache with a total maximum value. This technique is demonstrated by the example shown in Table 1. In the example, the fragments are sorted by value/size ratio. If the client's cache could hold 250 pages, then the maximal cached value would be obtained by caching fragments A and B (i.e., a total value of 800 in this case), and only these two fragments would be considered candidates by Reference-Counting. Likewise, if the client's cache could hold 300 pages, then the maximal cache value (900 in this case) would be obtained by caching A, B and half of C.[3] Thus, A, B, and C would be potential candidates.

3.3.3 The Profitable Policy

The second history-based policy we study, called Profitable, attempts to more closely approach the ideal algorithm described in Section 3.1 by directly estimating the ROI of fragments and taking into account the cost of investment. With the Profitable policy, $C_t(j)$, the component of equation 1 that is added to the value of fragment j for query t, is computed to be the improvement (in cost or response time, depending on what is being optimized) of running query t with j cached at the client, vs. running query t without j cached.

More specifically, $C_t(j)$ is calculated as follows: Using the optimizer's cost model, the cost (or response time) of executing query t is estimated assuming that the entire database is cached at the client. Again using the optimizer's cost model, the cost of executing the query is estimated assuming that all of the database *except* fragment j is cached at the client. $C_t(j)$ is computed as the difference between those two costs. If fragment j is not used in query t, $C_t(j)$ is set to 0.

Given this calculation of $C_t(j)$, the value of a fragment computed by equation 1 is thus the sum of the improvement that would have resulted from having fragment j cached for all past queries at the client, weighted by the aging factor α. This value is adjusted (reduced) to account for invalidations due to updates as described in Section 3.3.1 and is then used as an estimate of the ROI for caching that fragment for *future* queries. With the Profitable policy a fragment will be considered a candidate for query $t+1$ only if the following two criteria are met:

1. Its value (as calculated by equation 1) is greater than 0 and it would be fully or partially kept in a cache with a total maximum value (as defined for the Reference-Counting policy above).

2. Its value is higher than the investment required to initiate its caching as a by-product of executing query $t+1$.

The investment required to initiate the caching of fragment j while executing query $t+1$ is determined as follows: The optimal plan for query $t+1$ is generated given the actual, correct state of the client's cache. Then, a plan for query $t+1$ is generated assuming that, in addition, fragment j is also fully cached. Again using the optimizer's cost model, the investment required to initiate the caching of fragment j is computed

[3] Note that this calculation assumes a uniform distribution of value in a fragment. Other distributions can be handled at the expense of complicating the algorithm.

as the difference in cost (or response time) of the two plans. If fragment j is already cached or is not used in the query, the two plans are identical and the investment is 0.

Obviously, there are many ways to model the value and investment of caching. We chose these definitions in order to model the fact that the long-term benefits of caching depend on the queries (and not on the current state of the cache) while the investment of caching depends only on the current query and the current state of the cache.

3.4 Influencing Query Optimization

While cost and benefit concerns arise in any caching or replication scenario, a novel aspect of cache investment is the interaction of such concerns with query optimization. Rather than having a distinct process or mechanism whose job it is to continually reassess and modify the global data placement, cache investment works by influencing the query optimizer to generate query plans that (in conjunction with normal caching) result in a data placement that has long term advantages — even if such plans may hurt the responsiveness of particular queries in the short term.

Cache investment policies *fire* when they determine that a change in cache contents should be made. Policy firing initiates the caching of the candidate fragments that are used in a query when the query is activated for execution at a client. One way to implement policy firing is to change the internals of the query optimizer so that it can be forced to place scans of all candidate fragments at the client. Such an approach, however, limits the flexibility of the optimizer, and can also be quite difficult to implement, requiring a detailed understanding of the optimizer internals. In our case, we already had an existing optimizer that was capable of generating hybrid-shipping plans, and we did not want to modify it. As a result, we have developed an alternative way to integrate cache investment policies with the query optimizer.

The approach we have adopted effects the firing of a policy by *fooling* the optimizer to believe that scans of candidate fragments are very cheap at the client. As described in [FJK96] before performing operator site selection, our query optimizer obtains information about the contents of the client's cache from the buffer manager. This information is used by the optimizer's cost model to determine how expensive it is to place a scan at the client. When a policy fires, it fools the optimizer by *patching* the information passed from the buffer manager to the optimizer in a way that makes the optimizer's cost model believe that all the candidate fragments are cached at the client. The optimizer then *tends* to generate plans that place the scans of such relations on the client. In this scheme, neither the optimizer's search strategy nor its cost model must be changed. Also, this approach does not reduce the flexibility of site selection in a distributed system, because it merely tries to influence the optimizer's decisions, rather than dictate them. This latter property can benefit the performance of a cache investment policy because, for example, this approach would never initiate the caching of a fragment if queries using the fragment could always be executed most efficiently at the server — even if the fragment is cached at the client.

3.5 Summary of the Policies

In the following, we briefly summarize the four cache investment policies. All policies generate a set of *candidate* fragments that *should* be cached at a client and *fire* to initiate the caching of candidate fragments. We described two simple static policies that are used as baselines in the study that follows. The Conservative policy never considers fragments to be candidates, so it never fires. The Optimistic policy considers all fragments to be candidates, so it fires for every query. These policies are essentially the traditional query-shipping and data-shipping execution policies. Examining the limitations of these policies shows where the longer-term approach of cache investment can pay off.

We then introduced two history-based policies, Reference-Counting and Profitable, in which the choice of candidates depends on the sizes of the fragments, the size of the cache, and on the past history of queries submitted at a client. The use of history enables these policies to better adapt to a client's workload. The Reference-Counting policy considers only the most frequently used fragments to be candidates and ignores the cost of investment. The Profitable policy calculates an expected ROI for each fragment and chooses as candidates those fragments who have the highest expected ROI and whose investment cost is less than their expected ROI. The use of history information allows these policies to make decisions based on a longer-term view of the costs and benefits of various data placements. The question between these two policies is to determine if and under what conditions the additional complexity of the Profitable policy is worthwhile.

Finally, it is important to reiterate that unlike traditional caching and replication approaches, cache investment is an *indirect* method for effecting data placement. That is, investment policies work by influencing the query optimizer to generate hybrid-shipping query plans that result in the desired placement of data. This approach allows cache investment to be integrated without changing the internals of the optimizer search strategy and allows caching decisions to take advantage of the optimizer's cost model.

4 Experimental Environment

To investigate the relative performance of the four cache investment policies, we extended the simulation environment used in a previous study of client-server query processing [FJK96]. Figure 1 shows the overall structure of the simulator; the server model, the network model, and most components of the client model (including the query optimizer) are identical with those described in [FJK96]. In the following, we describe these models and specify the query workloads used in this study. Furthermore, some details of the query optimizer and its cost model are given. The results of the performance experiments are then presented in Section 5.

4.1 Simulation Environment

4.1.1 System Model

The client model consists of modules that generate, optimize, and execute queries, and that manage the client's main memory and disk that are both used to execute query operators and to cache data across

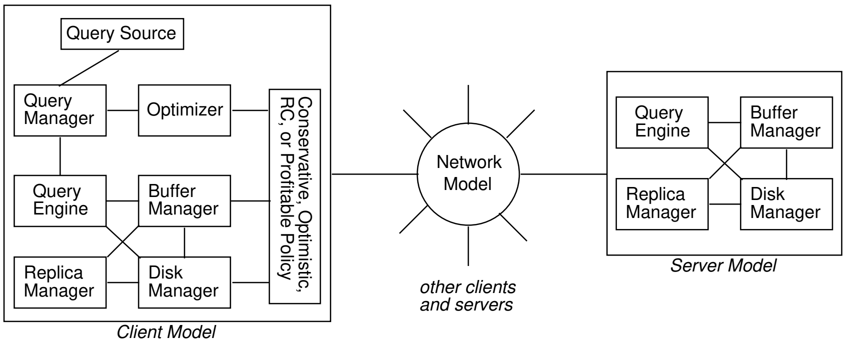

Figure 1: Simulation Model

queries. Queries are submitted by the *Query Source* one-at-a-time. As soon as a query has been completed, the next query is submitted with no think-time so that one query is always active at every client. For every query, a query plan is generated by the *Optimizer*. At this point, one of the four cache investment policies takes effect by passing information about the contents of the client's main-memory and disk cache to the optimizer; if the policy fires, this information is patched as described in Section 3.4. The query is then executed according to the optimized query plan: some of the operators are executed on the client and others on servers. The execution of an operator is simulated by the *Query Engine* which is based on an iterator model (similar to that of Volcano [Gra93]) and which provides implementations for scan, select, project, join, and network operators. Although the query engine includes several join methods, only hash joins are used in this study. Network operators effect communication between operators that run on different sites. The *Buffer Manager* allocates memory to operators; if necessary, the *Buffer Manager* schedules the query operators to avoid thrashing. Using an LRU replacement policy, the *Buffer Manager* also decides which pages of fragments are kept in the client's main-memory cache. When a page is replaced from the client's main memory, it is *demoted* to the client's disk cache which is managed by the *Disk Manager* using a FIFO replacement policy as devised in [FCL93]. In some experiments, pages are invalidated in the client's cache to model cache consistency maintenance for updates. These invalidations are effected by the *Replica Manager*. The *Replica Manager* does not model the specifics of any particular cache consistency protocol; it is intended to model the effects of invalidations on the performance of a cache investment policy that can be observed no matter what cache consistency policy is used.

The *Query Engine*, the *Buffer Manager*, and the *Disk Manager* of a server are identical with those of a client. Since no queries are submitted at servers, the server model does not have a *Query Source* or an *Optimizer*. Data owned by other servers can be cached at a server (following [CDF+94]); but to concentrate on the effects of client-side caching, most experiments are carried out with a single server that owns the whole database. The role of the *Replica Manager* in the server model, therefore, is primarily to trigger the invalidation of stale copies of pages cached at clients (rather than invalidate stale copies of data in a server's cache).

The network is modeled simply as a FIFO queue with a specified bandwidth; 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. Fragments and the results of (sub-) queries are always sent across the network a-page-at-a-time.

4.1.2 Model Parameters and Default Settings

Table 2 shows the most important simulation parameters and their default settings used in this study. It lists the parameters that model the CPU costs to execute hash joins (*Compare, HashInst, MoveInst*), to send and receive messages (*MsgInst, PerSizeMI*), and to read a page from disk (*DiskInst*). In addition, it lists the parameters that model the resources of a system. In this study, systems with up to 10 clients and 10 servers are used. The bandwidth of the network is set to 100 Mbit/sec; in some experiments, however, we measure the volume of data sent across the network to study the performance of the cache investment policies if communication is expensive. Every site (client and server) has a CPU whose speed is specified by the *Mips* parameter and a disk. The disk model is very detailed, including features such as an elevator scheduling policy, a controller cache, and read-ahead prefetching. The disk model parameters are not shown in Table 2, but they are set for clients and servers in the same way. They are based on a Fujitsu M2266 disk drive with an average performance of 3.5 msec per page for sequential I/O and 11.8 msec per page for random I/O; the size of a page is 4096 bytes. The client disk cache is varied from 0% to 40% of the size of the active database (i.e., that data that is within the access range of the client), and the main memories of clients and servers are varied from 2% to 40%. In any configuration, every site is given sufficient main memory to allow join processing at every site.

Parameter	Value	Description
NumClients	1 or 10	number of clients
NumServer	1 or 10	number of servers
Mips	50	CPU speed of a site (10^6 inst/sec)
NumDisks	1	number of disks on a site
ClMemory	2 - 40	client's main memory (% of database)
ClDiskCache	0 - 40	client's disk cache (% of database)
ServMemory	2 or 40	server's main memory (% of database)
NetBw	100	network bandwidth (Mbit/sec)
PageSize	4096	size of one data page (in bytes)
Compare	2	instructions to apply a predicate
HashInst	9	instructions to hash a tuple
MoveInst	1	instructions to copy 4 bytes
MsgInst	20000	instructions to send or receive a message
PerSizeMI	12000	instructions to send or receive 4096 bytes
DiskInst	5000	instructions to read a page from disk

Table 2: Simulator Model Parameters and Default Settings

4.2 Query Optimization

In the model, queries are optimized fully at run-time. Query plans are produced by a randomized query optimizer. Our randomized query optimizer is based on the approach described in [IK90] extended to carry out site selection in addition to other decisions such as join ordering. The optimizer can be configured in two different ways: (1) to minimize the *cost* of a query based on estimates à la [ML86], or (2) to minimize the *response time* according to the model of [GHK92]. In both modes, the cost-model parameters are set depending on the client-server configuration; for example, the cost model assumes that operations at a server are more expensive in a system with 10 clients than in a system with one client due to the expected higher load on the server. Furthermore, the cost model uses information about the contents of the clients' and servers' memories and disk caches; this information is refreshed and possibly *patched* (as described in Section 3.4) by a cache investment policy every time a query is optimized.

4.3 Workload Specification

The database used in this study consists of 100 relations. Each relation has 10,000 tuples of 100 bytes (1 MB); that is, the whole database has 100 MB.[4] For simplicity, the relations are not partitioned (i.e., every fragment is a whole relation), and no indexes are defined. Taking indexes into account, however, is an important point for future work. In most experiments, the whole database is stored on a single server; in experiments with 10 servers, each server stores exactly 10 relations.

The workload consists of a sequence of two-way join queries. The following two kinds of queries are used:

NoSel: A functional join in which every tuple of one relation matches exactly one tuple of the other relation. The result has 10,000 tuples of 100 bytes (1 MB).

HiSel: A functional join as in the *NoSel* query; but, with a 10% selectivity predicate applied to the inner relation of the join. The result of a *HiSel* query has 1,000 tuples of 100 bytes (100 KB).

For *NoSel* queries, the investment required to initiate the caching of relations is low: for example, if one relation of a *NoSel* query is already cached at the client, the caching of the other relation can be initiated with no extra communication cost (i.e., the investment is 0). For *HiSel* queries, on the other hand, the investment required to initiate the caching of relations is relatively high: the communication cost to move both relations to the client is 2 MB, and the benefit that can be achieved if both relations are cached is only 100 KB per query.

The relations participating in a query are chosen using two different distributions:

Uniform: Every relation is used with the same probability (2%) in a query.

Zipf: According to a Zipf distribution, some relations are used in more queries than others. At every client, a random permutation of the 100 relations is generated; different clients have different permutations to model that every client has individual preferences. The first relation of this permutation is used with the highest probability (approximately 38% per query), the second relation with the second highest probability (19%) and so on. We say that the relations at the beginning of the permutation are *hot* and the relations at the end are *cold*.

[4] The database and relation sizes are kept small in order to achieve acceptable simulation times. It is important to note (as demonstrated in [CFZ94]) that rather than the absolute sizes of the cache or data, it is their ratio that is important to measure the effectiveness of caching. We vary this ratio as part of our experimental framework.

For the Uniform distribution, the ROI of caching a relation is small because after the relation has been moved to the client, it is not likely to be used in one of the next queries at the client. For the Zipf distribution, the ROI of caching *hot* relations is very high because these relations are used in many future queries at the client. The four workloads used in this study are combinations of the two query types with the two access distributions. In terms of the investment and ROI of caching, the four workloads are characterized in Table 3.

	NoSel	*HiSel*
Uniform	low investment low ROI	high investment low ROI
Zipf	low investment high ROI	high investment high ROI

Table 3: Workload Classification

5 Performance Experiments and Results

5.1 Plan of Attack

In this section, we use the simulator and the workloads described previously to investigate the tradeoffs of the cache investment policies described in Section 3. We first compare the communication costs and throughput of the static and history-based policies in cases where the two history-based polices perform similarly. The goal of this comparison is to determine to what extent, and in which cases the flexibility provided by considering history can be helpful. We then focus on the differences between the two history-based policies. These differences are examined in several contexts, including heterogeneous servers and in the presence of invalidations due to updates. After comparing the run-time performance of the policies, their compile-time costs are examined in Section 5.6.

The experiments are conducted in the following manner: Initially, all client caches are empty. A stream of queries (e.g., *Uniform-NoSel*) is executed at every client. For every data point, at least 800 queries are executed to make sure that the 90% confidence intervals for all results are within $\pm 5\%$; the confidence intervals are computed using batch means [LC79]. Except where noted, the aging parameter α is set to 1 for the history-based policies, so that all queries in a history are given the same weight.[5] The sensitivity of the history-based policies to the value of α issue is addressed directly in Section 5.7.

5.2 Experiment 1: Communication Cost

In the first set of experiments, we examine the communication requirements of the various policies. These experiments are intended to model conditions where communication is slow or expensive, for example, in environments such as the Internet or mobile computing. For these experiments the optimizer is configured to minimize the cost of queries (rather than the response time). All experiments are carried out in a system with one client and one server that stores a copy of all relations of the database. In these experiments

[5] Since the workloads used in this section do not change over time, the sensitivity of α is not an issue here.

we vary the size of the client's cache; because only communication costs are measured, it does not matter whether data are cached in the client's main memory or disk. We fix the client's main-memory to 2% of the size of the database (i.e., two relations) and vary the size of the disk cache. The size of the server's memory was also set to 2%, but this has no effect on the communication costs.

Figures 2 through 5 show the average volume of data sent across the network per query for the various policies under each of the four workloads described in Section 4.3. In general, the results show that as expected, each of the static policies performs well for some situations but poorly for others, while the more flexible history-based policies have reasonable communications behavior across all four workloads, and in many cases are able to beat both static policies.

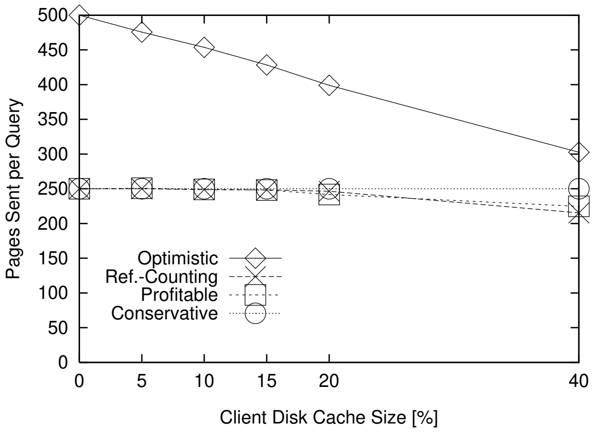

Figure 2: Pages Sent per Query
Uniform, NoSel, 2% Client Memory, Vary Disk Cache

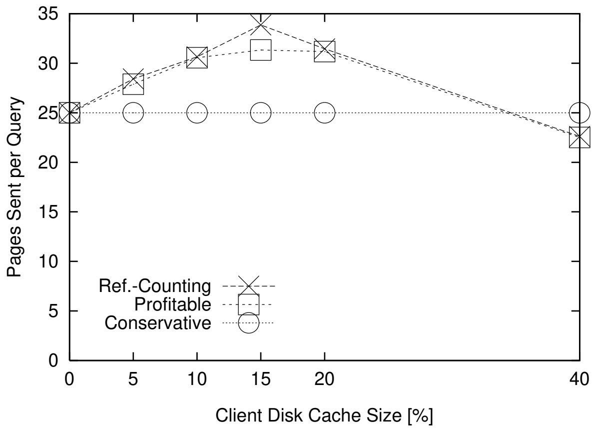

Figure 3: Pages Sent per Query
Uniform, HiSel, 2% Client Memory, Vary Disk Cache

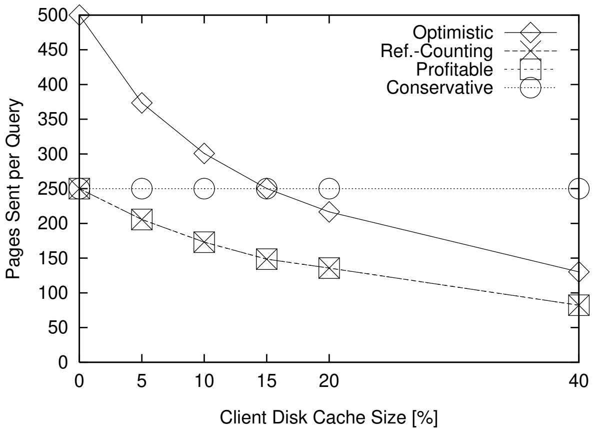

Figure 4: Pages Sent per Query
Zipf, NoSel, 2% Client Memory, Vary Disk Cache

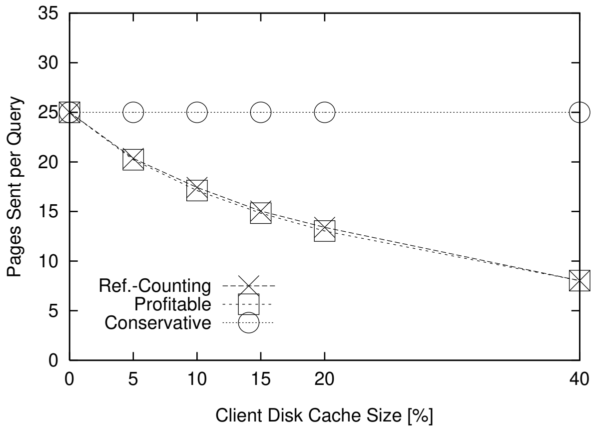

Figure 5: Pages Sent per Query
Zipf, HiSel, 2% Client Memory, Vary Disk Cache

In terms of the individual policies, it can first be seen that in all four workloads, the volume of data sent using the Conservative policy is independent of the size of the client's cache because all queries are executed at the server; its communication costs are solely determined by the size of the query result. The query result sizes are 1 MB (250 pages), and 100 KB (25 pages) for the *NoSel* and *HiSel* cases respectively.

In contrast, the Optimistic policy executes all queries at the client so its communication requirements are independent of the selectivities of the queries (i.e., they are identical for the *NoSel* and *HiSel* cases). In fact, because the Optimistic policy cannot exploit the communication benefits of small result sizes, it sends an order of magnitude more data than the other polices under the *HiSel* workloads. For this reason, the curves for Optimistic are not shown in Figures 3 and 5.

For Optimistic, the communication costs depend solely on the amount of base data accessed by a query that is not cache-resident at the client prior to the query execution. If no data is cached, then both relations of a join (500 pages) must be shipped. In Figure 2 it can be seen that under a *Uniform* workload, where all relations are used with equal probability, Optimistic's communication costs decrease linearly with the size of the client's disk cache. For the *Zipf* workloads such as *Zipf-NoSel* (Figure 4), Optimistic is able to make better use of client caches by caching *hotter* relations, so its communication costs are lower than in the uniform workloads. In the case of the *Zipf-NoSel* workload, Optimistic crosses Conservative at a cache size of 15%. At this point, the ROI of caching *hot* relations is approximately the same as the *loss* of initiating the caching of *cold* relations when the Optimistic policy is used.

Turning to the history-based policies, the Reference-Counting and Profitable policies show almost the same behavior in these experiments because in general, they tend to choose the same relations as candidates; The frequency of access and the ROI of caching fragments is roughly correlated in these cases. For the Uniform workload, with small caches, the history-based policies carry out most joins at the server and thus, behave similarly to the Conservative policy. Some caching is performed, however, as can be seen by comparing the *Uniform-NoSel* case (Figure 2) with the *Uniform-HiSel* case (Figure 3). In the high-selectivity case, for most of the cache sizes shown, the caching done by these policies results in a slight increase in communication. With small caches, Uniform workload, and a small result size, the ROI of caching is less than the investment cost — fragments do not stay in the cache long enough to repay the investment. As the cache size is increased, however, the fragments remain cache-resident longer which results in a slight win in this workload (e.g., with a 40% cache).

For the Zipf workloads (Figures 5 and 4), the history-based policies carry out queries on the hottest relations at the client, while queries on the *cold* relations are processed at the server. This behavior has two benefits: first, queries that access only the hottest relations are executed with no communication costs; and second, the hottest relations remain cache-resident longer because they don't compete for client cache space with the colder relations. The results is that the history-based polices have significantly lower communication costs than both static policies here.

5.3 Experiment 2: Throughput, Single Server

In the previous experiment, the volume of data sent across the network was measured in order to assess the tradeoffs for network-bound query processing. In this experiment, we study the performance of the policies when the network is not the bottleneck (i.e., 100 Mbit/sec network bandwidth). In this case, the optimizer is configured to minimize response time (rather than cost) and we use the throughput of the system in queries per second as our main performance metric.[6]

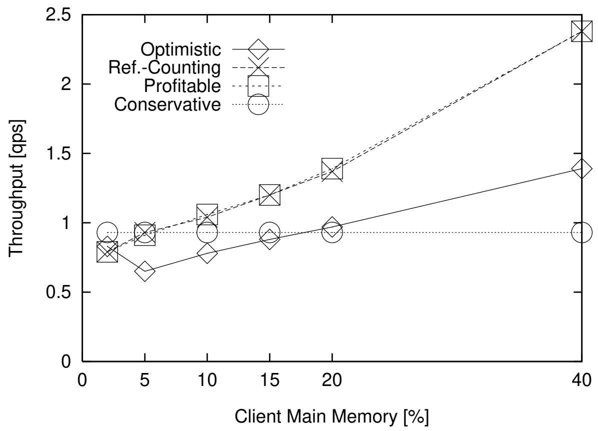

Figure 6: Throughput, 1 Client, 1 Server
Zipf, NoSel, Vary Client Memory, No Disk Cache

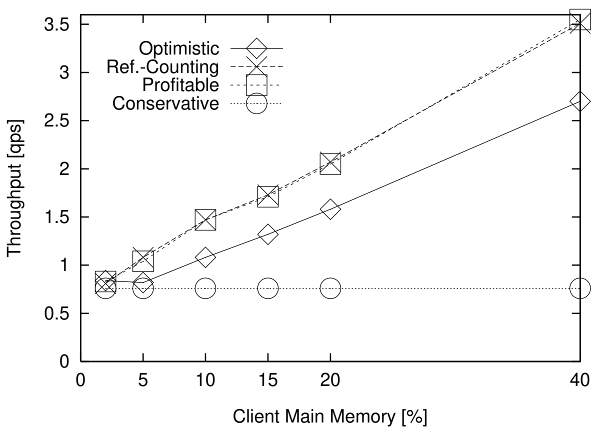

Figure 7: Throughput, 10 Clients, 1 Server
Zipf, NoSel, Vary Client Memory, No Disk Cache

Figure 6 shows the throughput of the four policies in a system with one client and one server under the *Zipf-NoSel* workload.[7] The size of the client's memory cache is varied from 2% to 40% of the size of the accessed data and there is no disk cache. As in the previous experiments, the history-based policies perform similarly, and for most cache sizes, are better than both of the static policies. Comparing the static policies, Optimistic shows better performance than Conservative for a cache size of more than 15% and worse performance for smaller caches. Even though the Optimistic policy uses both the client's and the server's main memory, it performs approximately the same total number of disk I/Os as the Conservative policy. This is because there is a large overlap between the client's and the server's memory: when a relation is loaded into the client's cache, this relation also subsequently remains in the server's main memory.[8] The difference in the performance of the two policies, therefore, is due to their different communication requirements as was shown in Figure 4. The history-based policies perform well in this case because they cache the *hottest* relations on the client and use all other relations on the server. In addition to lowering communication costs (as was seen in Figure 4), this behavior also reduces disk I/O by limiting the overlap of the contents of the client cache and the server's main memory.

Figure 7 shows the throughput of the policies under the same settings as Figure 6, but with 10 clients connected to the server instead of one. In this case, the server becomes a bottleneck, so the ROI of caching is much higher. As a result, the Conservative policy, which does all query processing at the server, performs the worst, and the history-based policies perform best here. The Optimistic policy performs somewhat worse than the history-based policies because it brings all relations to the client, and as a result, *cold* relations sometimes push *hot* relations out of the cache.

[6] Since we are using a closed model, the inverse relationship between throughput and response time makes either metric sufficient.

[7] We show results only for the *Zipf-NoSel* workload here, as the tradeoffs seen for the other cases are similar. In particular, the cheap communication makes the investment required to initiate caching for the *HiSel* and *NoSel* workloads nearly identical in terms of response time.

[8] The performance of the Optimistic policy in this case could be significantly improved using *global memory management* techniques devised in [FCL92] to reduce the overlap between the client's and the server's main memory.

5.4 Experiment 3: Throughput, Heterogeneous Servers

In all of the experiments so far, the two history-based policies showed roughly the same performance. This was because in those experiments the relations that were used most frequently also had the highest ROI for caching. If the database is distributed over several heterogeneous servers, however, this property may not always hold. In such an environment it is often better to cache data from a slow or far away server even if it is used in fewer queries. This effect is demonstrated in Figure 8 which shows the throughput of the policies under the *Uniform-NoSel* workload in a system with a diskless client and 10 servers; the client's memory is varied along the x-axis. Exactly 10 relations reside on every server. 5 Servers are *fast*; they have 40 MB of main memory so that the 10 relations stored on a fast server are always main-memory resident. The 5 other servers are *slow*; they have a small memory of 2 MB so that reading a relation from a slow server usually requires disk I/O.

The tradeoffs between the Conservative and the Optimistic policies have been demonstrated in previous experiments. The key result shown in Figure 8 is that the Profitable policy significantly outperforms the Reference-Counting policy when the client's cache becomes large. The Reference-Counting policy always initiates the caching of relations that are used in many queries; but, under a Uniform workload, the frequency with which relations are used is not a very good criterion for caching decisions. A better approach is to cache the relations of slow servers. This is done by the Profitable policy which is based on the abstract notion of caching relations that promise the highest benefits of caching.

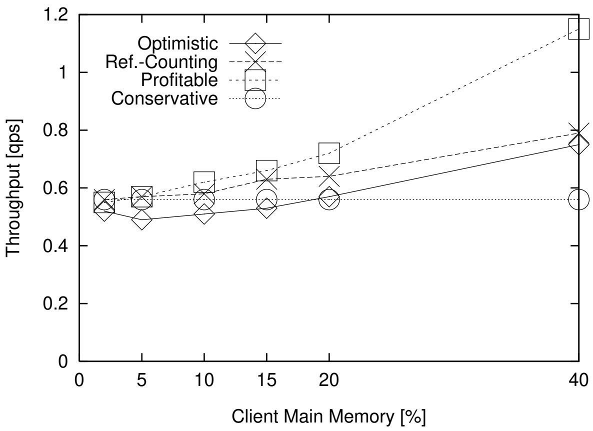

Figure 8: Throughput, 1 Client, 10 Servers
Uniform, NoSel, Vary Main Memory, No Disk Cache

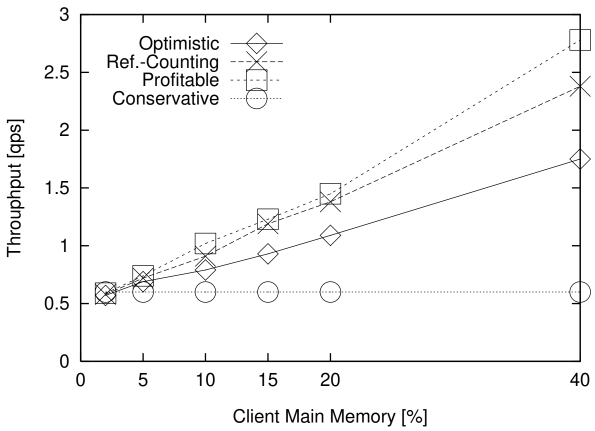

Figure 9: Throughput, 1 Client, 10 Servers
Zipf, NoSel, Vary Main Memory, No Disk Cache

Figure 9 shows the throughput in the same system setup under the *Zipf-NoSel* workload. Again, caching relations of slow servers is better than caching relations of fast servers. In a Zipf workload, however, the frequency in which relations are used is a good criterion too, and therefore, the Reference-Counting policy loses only slightly to the Profitable policy.

5.5 Experiment 4: Impact of Invalidations

Another reason that the performance of the two history-based policies may differ is because of invalidations of cached pages due to updates. Invalidations decrease the ROI of cached fragments, and hence, a policy that does not take them into account (such as Reference Counting) may suffer. In this experiment we investigate the impact of invalidations. Invalidations are modeled in the following way: pages are continuously updated at the server using a uniform distribution; after the execution of a query, the client's *Replica Manager* asks the server which pages of the database have been updated while the query was executed.[9] This information is used to invalidate stale copies of pages cached at the client and to adjust the *values* of relations for the history-based policies (Section 3.3.1). In this experiment, the size of the client's disk cache is fixed at 15%, and the *invalidation rate* is varied. The invalidation rate specifies the average portion of the database that is updated at the server during the execution of a query. Only the results for the Zipf workloads are shown; the results under the Uniform workloads show basically the same effects.

Figure 10 shows the communication costs of the four policies under the *Zipf-NoSel* workload. The communication costs of the Conservative policy are independent of the invalidation rate because the Conservative policy never initiates the caching of data. The Optimistic policy, on the other hand, initiates the caching of data, and therefore, its communication costs can be impacted by invalidations. At low invalidation rates, the two static polices perform equivalently (as was seen for the 15% point in Figure 4). At a higher invalidation rate, however, the ROI of caching a relation (even a hot one) decreases so the communication costs of Optimistic increase.

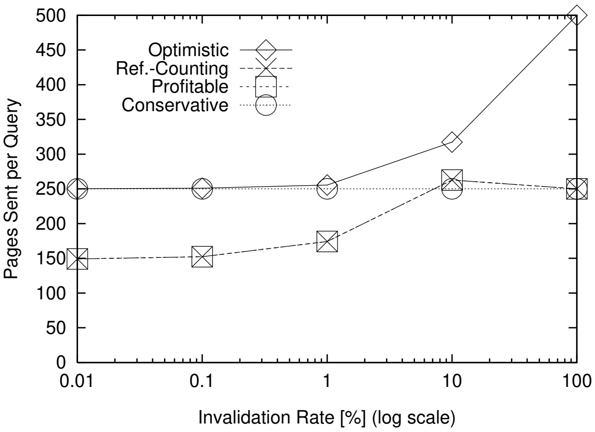

Figure 10: Pages Sent, Vary Inv. Rate
Zipf, NoSel, 2% Client Memory, 15% Disk Cache

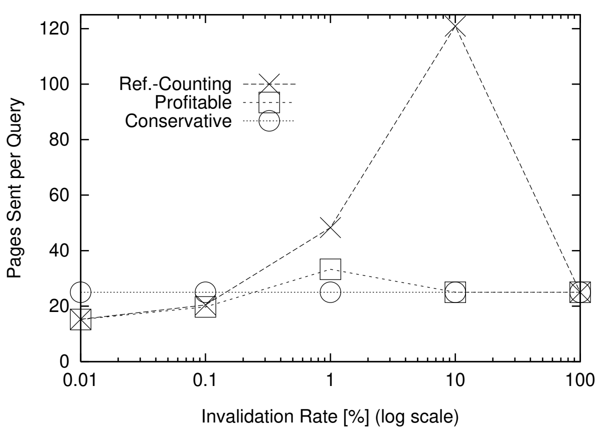

Figure 11: Pages Sent, Vary Inv. Rate
Zipf, HiSel, 2% Client Memory, 15% Disk Cache

The Reference-Counting and Profitable policies outperform both static policies at low invalidation rates as was seen in the no update case described previously. For a 10% invalidation rate, the ROI of caching *hot* relations is approximately the same as the investment required to continuously re-initiate the caching of the pages of these relations, so the history-based policies have almost the same overall communication costs as

[9] We do not model the overhead (i.e., control messages) of this interaction with the server in order to concentrate on the effects of invalidations and because we do not model a specific cache consistency policy.

the Conservative policy. When the invalidation rate is increased to 100% the history-based policies no longer initiate caching (because all fragment values become 0), so they have the same costs as Conservative.

Figure 11 shows the results for the *Zipf-HiSel* workload.[10] The Reference-Counting policy has low communication costs if the invalidation rate is less than 1%; but these increase steeply with the invalidation rate. At 100% invalidation, as noted earlier, the history-based policies are identical to the Conservative policy. The reason that Reference-Counting has such high communication costs when the invalidation rate is increased here, is because in the *HiSel* workload the cost of investment is very high, but this cost is completely ignored by Reference-Counting. In contrast, the Profitable policy, which does consider investment cost, curtails caching at a much lower invalidation rate, and so it has much lower communication costs than Reference-Counting here.[11]

5.6 Experiment 5: CPU Costs of Query Optimization

The focus of the previous experiments was on the quality of query plans. The time to optimize a query was not taken into account because only simple two-way join queries were used, and the time to optimize these queries was negligible compared to the time to execute them. For complex queries, however, the optimization time can be significant and should not be ignored. In this section, therefore, we examine the average CPU time consumed to optimize a query in a system with 1 client and 1 server; these times were obtained using the Unix *getrusage* command while running our query optimizer on a Sun Sparcstation 5.

	Optimistic	Ref.-Counting	Profitable	Conservative
2-Way Join	2.6	2.7	13.4	1.3
5-Way Join	680.8	599.2	2556.9	533.8

Table 4: CPU Time to Optimize a Query (in msecs)
2 and 5-Way Joins, 2% Client Memory, 10 MB Disk Cache, 1 Client, 1 Server

There are two ways in which a cache investment policy influences the cost of query optimization: (1) a policy can induce overhead, say, to maintain a history of queries; and (2) because of the placement of scan operators, the size of the search space can be policy-dependent. As shown in Table 4, the optimization time is always the smallest for the Conservative policy because it has no overhead and always has the smallest search space: all scan operators are placed at the server (i.e., plans with scans at the client are not considered). Like the Conservative policy, there is (almost) no overhead required to effect the Optimistic policy; but since the Optimistic policy makes the optimizer believe that the whole database is cached at the client (Section 3.4), the search space is much larger than for the Conservative policy: using the Optimistic policy, a scan can be placed at the client or at the server. Obviously, the Reference-Counting policy requires some overhead to maintain statistics and to identify the fragments that are candidates; but compared to the cost of finding the best query plan, this overhead is very small. For 5-way joins, query optimization is

[10] The communication costs of the Optimistic policy are not shown; they are the same as under the *Zipf-NoSel* workload (Figure 4), i.e., they are an order of magnitude higher than Conservative here.
[11] Since the cost-benefit calculations of the Profitable policy are not always perfect it is slightly outperformed by the Conservative policy for an invalidation rate of 1%.

actually faster for the Reference-Counting than for the Optimistic policy because Reference-Counting has a smaller search space: only scans of relations that are cached or that are candidates can be placed at the client. The highest optimization times are required for the Profitable policy. This is because several plans must be generated for every query in order to compute the benefits and investments of caching the relations used in the query. In this paper, these plans are generated using the same join order and only repeating the optimizer's site selection phase; nevertheless, the optimization times in Table 4 are 5 and 10 times higher than for the Conservative policy.

5.7 Experiment 6: Tuning the History-Based Policies

The experiments presented in Sections 5.2 through 5.5 showed that in most situations, the Reference-Counting and the Profitable policies show better performance than the Optimistic and Conservative policies. In all the previous experiments, the aging parameter α for the Reference-Counting and Profitable policies was set to 1. Tuning the history-based policies for the previous experiments was not required because the characteristics of a client's workload never changed. The setting of α, however, can significantly influence the performance of a history-based policy if the workload of a client changes. In this section, we examine the speed with which the Reference-Counting and Profitable policies adapt to such changes in the workload under different settings of α. Changes in the workload are modeled by dividing the stream of queries into *sessions*. Within a session, relations are accessed according to a Zipf distribution; however, different relations are *hot* in different sessions.

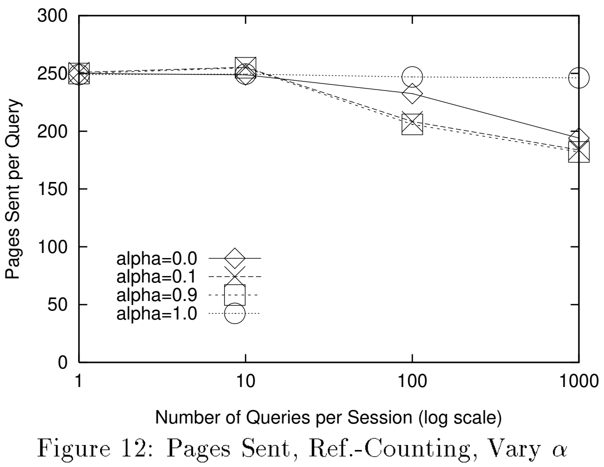

Figure 12: Pages Sent, Ref.-Counting, Vary α
Vary Sessions, NoSel, 2% Client Mem., 10% Disk Cache

Figure 12 shows the volume of data sent across the network for the Reference-Counting policy under various settings of α depending on the number of queries per session.[12] The more queries per session, the higher the ROI of caching the *hot* relations for that session. Setting $\alpha = 1$, however, shows high communication costs independent of the number of queries per session; for $\alpha = 1$, all queries of a history are given the same weight, and therefore, the caching of the *hot* relations of a new session is not initiated because

[12] The results for the Profitable policy are not shown because they are almost identical.

the *hot* relations of previous sessions retain their high values. In the other extreme, for $\alpha = 0$, a history of only one query is kept. In this case, the caching of a relation is initiated if it is used in two consecutive queries. As shown in Figure 12, this allows the policy to adapt to changes under a Zipf distribution because the *hottest* relations are often used in two consecutive queries, and as a result, $\alpha = 0$ shows lower communication costs than $\alpha = 1$ for more than 100 queries per session. The policies, however, adapt most quickly if α is set to 0.1 or 0.9; therefore, these settings show the lowest communication costs in Figure 12 for more than 100 queries per session. Actually, in all our experiments, we observed the phenomenon that the setting of α is fairly insensitive in the range of $0 < \alpha < 1$ (i.e., any value except 0 or 1) and that the policies adapt most quickly to changes in the workload if α is set within this interval.

6 Related Work

As stated in the introduction, caching and replication are complementary concepts but serve different purposes and different techniques and models have been used to support them. Nevertheless, there are some similarities in the design of some replication algorithms and the policies studied in this work. The design of the Reference-Counting policy is similar to some of the ideas of the dynamic replication algorithms presented in [WJ92b, WJ92a, AZ93]. These algorithms also record how often an object is read at a site and how often it is modified; but they are specifically designed for systems in which all operators of a query are executed at the client and updates are *propagated* to all replicas (rather than employing an invalidation-based protocol). Like the Profitable policy, replication in Mariposa [SAL+96, SAB+96] tries to estimate the costs and benefits of establishing a copy at a site; Mariposa's model, however, is intended to load balance a system, and it is not integrated into query optimization so that it does not address the circular dependency between caching and site selection.

More recently, several researchers have investigated the use of cost and benefit calculations for determining when to keep or replace cached data. WATCHMAN [SSV96] is a cache manager for data warehousing. It retains the answers to queries in order to avoid having to re-evaluate them. A *profit* metric is used to determine which results should be cache-resident. A set of related algorithms for distributed caching are proposed and studied in [SW97]. These algorithms use estimates of cost and benefit to determine a good placement of (possibly) replicated items in a distributed systems. Both of these papers propose history-based methods for determining the value of caching data. Neither of them, however investigate the interaction of caching and query optimization as arises in the *hybrid-shipping* architecture that is addressed in this paper.

Prefetching is another related area that has been studied extensively. Like the policies proposed in this paper, prefetching invests in the cache resource in order to make future accesses to the database more efficient. Prefetching, however, is integrated into the running-time system of a site, and it is intended to reduce the running time of an operation once it has begun; a cache investment policy, on the other hand, takes effect during query optimization when the execution of an operation is planned. Nevertheless, it might be possible to find new, better cache investment policies by adapting some of the techniques that were designed for prefetching to predict the future behavior of a client (e.g., [PZ91, CKV93]).

There has, of course, been a great deal of related work in the area of caching. In [FJK96], we studied how a query could be executed most efficiently given the contents of a client's cache. In that study, however, every query was optimized individually, and the impact of operator placement on caching was not taken into account. In [RK86, CR94, SJGP90, KB94, DFJ+96], caching was integrated into a query processing environment by caching the results of queries. Caching query results is still an open research topic by itself and, therefore, is not directly addressed here. The policies presented in this paper, however aim to control which data is cached at a site after the execution of a query, and therefore, should be of use in any caching system that supports queries.

7 Conclusions

We began this paper by identifying the circular dependency between caching and query optimization. To initiate the caching of data at a client, the query optimizer must sometimes produce a sub-optimal plan for a query that uses the data. Generating a sub-optimal plan can be a good investment if the data are used in many subsequent queries; but it can also decrease the value of the cache by flooding the client's cache with *cold* data and replacing *hotter* data that are used more often in future queries.

This paper presented and evaluated four policies which determine when and for which data the investment required to initiate caching should be paid, and it showed how these policies can be integrated into a system without changing basic components such as the query optimizer's search strategy, the query engine, or the buffer manager. The Conservative and the Optimistic policies mark extreme points of the design space and correspond to the ways distributed database systems have traditionally been built. Independent of the query workload of a client, the Conservative policy is *never* and the Optimistic policy is *always* eager to initiate the caching of data. Both policies make *static* assumptions about the investment and the return on investment (ROI) of caching data, and therefore, show poor performance if the query workload does not match these assumptions. Needless to say, database systems architectures based solely on one of these approaches will suffer from similar problems.

The Reference-Counting and Profitable policies take a longer-term, *dynamic* approach: they maintain statistics about past queries at a client and try to adapt to the client's query workload. In the performance experiments, they showed significantly better performance than the static policies in many situations. The Reference-Counting policy was motivated by replacement policies for buffer management (e.g., LRU, LFU) that record when and/or in how many queries data are used in order to predict which data are likely to be used in future queries. The Profitable policy was designed to approximate an ideal policy by estimating the costs and benefits of caching based on a history of past queries. It was not possible to identify a clear winner between the Reference-Counting and the Profitable policies: systems with heterogeneous servers or systems that are designed to process many update transactions should employ the Profitable policy; other systems should employ the Reference-Counting policy because it is easier to implement and has significantly less overhead.

There are many directions in which this work can be extended. One important area for future research is the caching of indexes. Recently, several algorithms for using and maintaining indexes in a caching environment have been proposed and studied in the context of the SHORE system [ZC97]. We plan to investigate the interaction of Cache Investment with these algorithms. Also, as mentioned previously, this study was carried out in an architecture in which only base data and no results of queries or sub-queries could be cached. If the results of sub-queries can be cached at a client, *all* operators (not only scans) could initiate caching. In separate work, we have proposed Semantic Caching [DFJ+96] as a compromise between physical caching and query result caching. Integrating Cache Investment with Semantic Caching is another promising direction. Finally, we intend to study how the policies studied in this work can be combined with techniques for global memory management such as those proposed in [FCL92].

Acknowledgments

We would like to thank Björn Thór Jónsson who designed and implemented the simulator used in the performance experiments. We would also like to thank Laurent Amsaleg for helping to improve the presentation of this paper.

References

[AZ93] S. Acharya and S. Zdonik. An efficient scheme for dynamic data replication. Cs-93-43, Dept. of Computer Science, Brown University, Providence, Rhode Island 02912, USA, September 1993.

[Bes95] A. Bestavros. Demand-based document dissemination to reduce traffic and balance load in distributed information systems. In *Proc. IEEE Symp. on Parallel and Distributed Processing*, San Antonio, TX, October 1995.

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

[CDF+94] M. Carey, D. DeWitt, M. Franklin, N. Hall, M. McAuliffe, J. Naughton, D. Schuh, M. Solomon, C. Tan, O. Tsatalos, S. White, and M. Zwilling. Shoring up persistent applications. In SIGMOD [SIG94], pages 383–394.

[CFZ94] M. Carey, M. Franklin, and M. Zaharioudakis. Fine-grained sharing in a page server OODBMS. In SIGMOD [SIG94], pages 359–370.

[CKV93] K. M. Curewitz, P. Krishnan, and J. S. Vitter. Practical prefetching via data compression. In *Proc. ACM SIGMOD Conf.*, pages 43–53, Washington, DC, USA, May 1993.

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

[CR94] C. Chen and N. Roussopoulos. The implementation and performance evaluation of the ADMS query optimizer: Integrating query result caching and matching. In *Advances in Database Technology – EDBT '94, Proceedings*, Cambridge, United Kingdom, March 1994.

[DFJ+96] S. Dar, M. Franklin, B. Jonsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In VLDB [VLD96], pages 330–341.

[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 VLDB Conf.*, pages 107–121, Brisbane, Australia, 1990.

[EH84] W. Effelsberg and T. Härder. Principles of database buffer management. *ACM Trans. on Database Systems*, 9(4):560–595, 1984.

[FCL92] M. J. Franklin, M. J. Carey, and M. Livny. Global memory management in client-server DBMS architectures. In *Proc. of the VLDB Conf.*, pages 596–609, Vancouver, Canada, 1992.

[FCL93] M. J. Franklin, M. J. Carey, and M. Livny. Local disk caching for client-server database systems. In *Proc. of the VLDB Conf.*, pages 543–554, Dublin, Ireland, 1993.

[FCL97] M. Franklin, M. Carey, and M. Livny. Transactional client-server cache consistency: Alternatives and performance. *ACM Trans. on Database Systems*, 1997. Accepted for Publication.

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

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

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

[IK90] Y. E. Ioannidis and Y. C. Kang. Randomized algorithms for optimizing large join queries. In SIGMOD [SIG90], pages 312–321.

[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.

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

[LC79] A. Law and J. Carson. A sequential procedure for determining the length of a steady-state simulation. *Operations Research*, 27(5):1011–1025, September 1979.

[LLOW91] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore database system. *Communications of the ACM*, 34(10):50–63, 1991.

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

[PZ91] M. L. Palmer and S. B. Zdonik. FIDO: A cache that learns to fetch. In *Proc. of the VLDB Conf.*, pages 255–264, Barcelona, September 1991.

[RK86] N. Roussopoulos and H. Kang. Principles and techniques in the design of ADMS. *IEEE Computer*, 19:19–25, December 1986.

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

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

[SIG90] *Proc. ACM SIGMOD Conf.*, Atlantic City, USA, April 1990.

[SIG94] *Proc. ACM SIGMOD Conf.*, Minneapolis, MI, USA, May 1994.

[SJGP90] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On rules, procedures, caching and views in data base systems. In SIGMOD [SIG90], pages 281–290.

[SSV96] P. Scheuermann, J. Shim, and R. Vingralek. Watchman: A data warehouse intelligent cache manager. In VLDB [VLD96].

[SW97] M. Sinnwell and G. Weikum. A cost-model-based online method for distribued caching. In *Proc. Intl. Conf. on Data Engineering*, Birmingham, U.K., 1997. IEEE.

[VLD96] *Proc. of the VLDB Conf.*, Mumbai, India, September 1996.

[WJ92a] O. Wolfson and S. Jajodia. An algorithm for dynamic data distribution. In *IEEE Workshop on Management of Replicated Data*, November 1992.

[WJ92b] O. Wolfson and S. Jajodia. Distributed algorithms for dynamic replication of data. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, pages 149–163, San Diego, June 1992.

[ZC97] M. Zaharioudakis and M. Carey. Highly concurrent cache consistency for indices in client-server database systems. In *Proc. ACM SIGMOD Conf.*, Tucson, AZ, May 1997.