Cache Investment: Integrating Query Optimization and Distributed Data Placement

Donald Kossmann
Technical University of Munich
and
Michael J. Franklin
University of California, Berkeley
and
Gerhard Drasch
WIG AG, Munich

Emerging distributed query processing systems support flexible execution strategies in which each query can be run using a combination of data-shipping and query-shipping. As in any distributed environment, these systems can obtain tremendous performance and availability benefits by employing dynamic data caching. When flexible execution and dynamic caching are combined, however, a circular dependency arises: Caching occurs as a by-product of query operator placement, but query operator placement decisions are based on (cached) data location. The practical impact of this dependency is that query optimization decisions that appear valid on a per-query basis can actually cause sub-optimal performance for all queries in the long run.

To address this problem, we have developed Cache Investment; a novel approach for integrating query optimization and data placement that looks beyond the performance of a single query. Cache Investment sometimes intentionally generates a "sub-optimal" plan for a particular query in the interest of effecting a better data placement for subsequent queries. Cache Investment can be integrated into a distributed database system without changing the internals of the query optimizer. In this paper, we propose Cache Investment mechanisms and policies and analyze their performance. The analysis uses results from both an implementation on the SHORE storage manager and a detailed simulation model. Our results show that Cache Investment can significantly improve the overall performance of a system and demonstrate the tradeoffs among various alternative policies.

Categories and Subject Descriptors: H.2.4 [**Information Systems**]: Database Management—*Relational Databases, Distributed Databases*

This work was partially supported by the NSF under grant IRI-94-09575, by the Office of Naval Research under contract number N66001-97-C8539 (DARPA order number F475), by Telcordia (formerly Bellcore), and by an IBM Shared University Research award. Donald Kossmann was supported in part by the Humboldt-Stiftung, UMIACS, and DFG Grant Ke 401/7-1.

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Cache Investment, Caching, Dynamic Data Placement, Query Optimization, Client-Server Database Systems, Data Shipping, Query Shipping

1. INTRODUCTION

Caching has emerged as a fundamental technique for ensuring high performance in distributed systems. Caching is an *opportunistic* form of data replication in which copies of data that are brought to a site by one query are retained at that site for possible use by subsequent queries. Caching is particularly important in large systems with many clients and servers because it reduces communication costs and off-loads shared server machines. As such, caching has been successfully integrated into many commercial and research database systems, data warehouses, and database application systems (e.g., SAP R/3). Ongoing research into distributed databases is aimed at developing more flexible models for query processing. In this paper we address the new challenges that arise when integrating caching with such advanced query processing systems.

1.1 Caching and Query Optimization

Recent activity in the development of distributed database architectures has focused on allowing the flexible and dynamic placement of query operators (e.g., selects, joins, aggregates, etc.) at various sites in the network [Jenq et al. 1990; Franklin et al. 1996; Stonebraker et al. 1996; Deßloch et al. 1998; Mayr and Seshadri 1999; Rodgriguez-Martinez and Roussopoulos 2000]. In such systems, query operators can be placed at sites in a way that minimizes expected communication costs, execution time, or other metrics. These decisions are based in large part on knowledge of which data is located at which sites. When caching is added to such a system, however, a complication arises because with caching, data location is determined as a by-product of query operator placement. That is, when data is brought to a site for processing, that data can be retained (i.e., cached) there for use by subsequent queries. This *circular dependency* between caching and query plan optimization has not been previously identified, but it has significant performance implications for advanced distributed database systems, as 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 which initially has no data cached. Both relations consist of 10,000 tuples of 100 bytes each (1 MB), and the result of the join is estimated to have 9,000 tuples of 100 bytes (0.9 MB). 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 query execution plan is to ship a copy of relation A to Server *II*, 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 place the join operator at the client and ship copies of relations A and B from the servers to the client at a communication cost of 2 MB.

In isolation, it would appear that the first plan is slightly preferable to the second.

Using the first plan, however, a subsequent query to join relations A and B with the same selectivity posed at the client 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 initial query been chosen instead, as that plan would have enabled A and B to be cached at the client.

Example 2 A highly-selective query on a very large relation C is submitted at a client that has no data cached. Assume that relation C is stored remotely on Server I. In this case 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. Furthermore, 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.

In contrast to the first example, this second example shows that in some cases, changing operator placement to effect caching can dramatically hurt the response time of the current as well as future queries. Taken together, these two examples demonstrate some of the potential advantages and pitfalls of combining caching and flexible query processing in general, and of the circular dependency between caching and query optimization in particular.

If the circular dependency is not addressed, a distributed query processing system will suffer from suboptimal performance and poor utilization of caching resources. The crux of the problem is that opportunistic caching of data at sites to which the data has been sent for query processing imposes an additional responsibility on the query optimizer: *operator site selection decisions made for one query have ramifications on the performance of subsequent queries.* As a result, the query optimization process must be extended to take a longer-term view of the impact of its decisions. Of course, query optimization is already one of the most complex and highly-tuned components of a database system — it is not likely that the developers of existing systems will be willing or able to rewrite the query optimizer from scratch. The challenge then, is to integrate the notions of data placement and distributed query optimization in an efficient and effective manner, without greatly impacting the architecture and complexity of the query optimization process.

1.2 Proposed Solution - Cache Investment

Our approach to this problem, called *Cache Investment*, is a novel technique for combining data placement and query optimization. Rather than requiring the creation of a new optimizer from scratch, Cache Investment is implemented as a module that sits outside of the query optimizer. This module influences the optimizer to sometimes make *suboptimal* operator site selection decisions for individual queries in order to effect a data placement that will be beneficial for subsequent queries. In other words, it causes the optimizer to *invest* resources during the execution of one query in order to benefit later queries.

In this paper, we describe how such a cache investment component can be built and integrated with the query optimizer without changing its basic components such as plan enumeration, search strategy, or cost model. In fact, our approach only *influences* the decisions made by the optimizer, rather than dictating them.

In this way, the optimizer is able to disregard the hints provided by cache investment when it deems the investment cost to be too high.

Another benefit of our Cache Investment approach is that it requires no changes to any components of the runtime query execution system. The same query operators for *scan, joins, group-bys*, etc. can be used, and once the data is cached, it can be managed using well-known buffer replacement policies such as LRU, LRU-k [O'Neil et al. 1993], or the policies described in [Effelsberg and Härder 1984].

Of course, caching and the related issues of replication and prefetching have been extensively studied in the literature. Cache investment builds on this previous work, but differs from it in that it carries out data placement decisions by influencing a database query optimizer. Previous techniques have either been static, driven by specific user requests (as in data shipping systems), or have used mechanisms that do not exploit existing query optimization functionality. Such solutions are at best, ignorant of query optimization decisions, and at worst, in conflict with them. Thus, the key contribution of our work is the development of an effective, integrated solution that aims to minimize the changes that must be made to existing system components. We discuss related work in detail in Section 7.

1.3 Overview of the Paper

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 describes cache investment. First, this section shows how the optimizer can be extended without changing its basic components. Then, two example cache investment policies are presented. We evaluate the two cache investment policies in an extensive performance study using the distributed database system SHORE [Carey et al. 1994] and a simulator as experimental platforms. As a baseline for our study, we use system architectures that correspond to the ways that existing state-of-the-art systems are built. Sections 4, 5, and 6 describe the results of the performance experiments. Section 7 discusses related work. Section 8 presents conclusions and suggestions for future work.

2. ARCHITECTURE AND ASSUMPTIONS

In this paper, our focus is 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, are also helpful for other distributed database architectures such as symmetric peer-to-peer systems in which every site can act as both a client and a server. For peer-to-peer systems, our techniques need to be adjusted because in such systems caching at one node can impact the execution of queries submitted at other nodes, too. Furthermore, we assume a *hybrid-shipping* client-server query processing model as described in [Franklin et al. 1996]. 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. We first, however, outline several assumptions we make about the type of caching supported by the system.

2.1 Cache Management Assumptions

In general, database caching can be done using *physical* or *logical* techniques. Physical caching is performed in terms of records, pages (i.e., blocks), or static partitions of base tables. Logical caching is performed in terms of query results or subsets of query results [Roussopoulos and Kang 1986; Chen and Roussopoulos 1994; Stonebraker et al. 1990; Keller and Basu 1994; Dar et al. 1996]). Physical caching is used by most distributed systems today, including object-oriented database systems, network file systems, and database application systems such as SAP R/3. In contrast, many aspects of logical caching remain the subject of active research. Thus, in our work here, we focus on physical caching. More specifically, we implement caching at the granularity of data and index pages.

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, updates cause invalidation messages to be sent to clients that have cached page copies. The new version of a page is shipped to a client only upon request. This is in contrast to 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 [Franklin et al. 1997; Zaharioudakis and Carey 1997]. Callback locking is a prominent example of an invalidation-based policy, and it is used in several client-server database systems; e.g., ObjectStore [Lamb et al. 1991] and SHORE [Carey et al. 1994]. As we will see, however, cache investment and the techniques we propose in this paper can also be applied to systems that use different cache consistency protocols.

In this paper we focus on a client-server architecture in which data is cached in the clients' main memories. We have also applied cache investment for disk caching, but we concentrate on the main-memory case here for ease of presentation. (See [Franklin and Kossmann 1997] for cache investment results with local disk caching.) Finally, it is important to note that cache investment does not impact the way replacement decisions for cached data are made. Rather, cache investment takes effect during query optimization and influences which data is brought to a client for possible caching. Within the cache, replacement victims can be chosen using a "standard" replacement policy such as LRU or LRU-k.

2.2 Hybrid-Shipping Query Processing

As described in [Franklin et al. 1996], the two key aspects of hybrid-shipping query processing are: 1) flexible site selection for query operators, and 2) 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 [Franklin et al. 1996], there are cases where the operators of a single query should be split among clients and servers.

At present, most commercial client-server database systems (e.g., Oracle 8) do not provide the flexibility to choose among these options. A flexible approach, however, has become increasingly popular in experimental systems such as ORION-2 [Jenq et al. 1990], KRISYS [Deßloch et al. 1998], Mariposa [Stonebraker et al. 1996], Predator [Mayr and Seshadri 1999], and MOCHA [Rodgriguez-Martinez and Roussopoulos 2000], and has been integrated into an extended version of the SHORE storage manager [Carey et al. 1994] as part of the work described here. In the commercial world, SAP R/3 is a prominent example of a system that has a hybrid-shipping architecture [Buck-Emden and Galimow 1996; Kemper et al. 1998].

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 [Carey and Lu 1986]. 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 [Carey and Lu 1986; Stonebraker et al. 1996; Franklin et al. 1996].

2.3 Hybrid-Shipping with Caching

In a hybrid-shipping architecture that supports physical caching, caching at a client is initiated by placing a *scan* operator of a query on the client. A scan takes a table or index as input and delivers a stream of tuples (or RIDs in the case of index scans) as output. When a scan is executed at a client, it uses any pages of the table or index that are cache-resident at the client — all other pages are *faulted in* from the 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 ignored and no new pages of the table or index can be cached at the client. An index scan at a client only brings those pages of the index and base table to the client that contain items that satisfy the query predicate; a table scan brings all pages of the base table to the client. With physical caching, no other operators (e.g., joins) are used to initiate caching, as the output of such operators are logical entities (i.e., a sub-query result). Cache investment for query and sub-query results is an interesting avenue for future work (see Section 7).

Of particular interest for cache investment is how the optimizer determines whether a scan operator should be executed at the client or at a server. In order to estimate the cost of a scan operator for a table at a client, the optimizer will ask the cache manager how many pages of that table are cached at the client. Correspondingly, the optimizer will ask for the number cached pages of the index and underlying table to estimate the cost of an index scan at the client. If large portions of a table are cached at the client, for example, then the optimizer will most likely place the scan operator at the client because such a scan operator is cheap. We say "most likely" because the ultimate decision where to execute scans

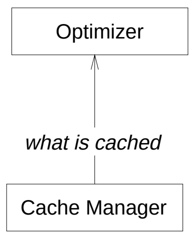

Fig. 1. Traditional Optimizer

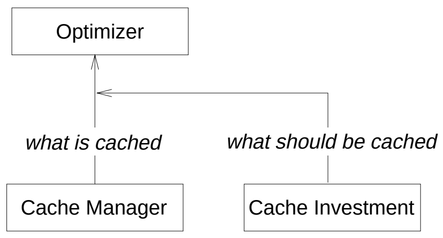

Fig. 2. Integrating Cache Investment

also depends on the (CPU) power and load of the client and server machines, the bandwidth of the network, and other operations of the query (e.g., joins).

On the other hand, if none of the pages of a table are cached, a traditional optimizer will always place the scan operator at a server that has a copy of the table. The reason is that executing a scan at a server and shipping the tuples in large blocks to the client (possibly after applying a query predicate at the server) is always cheaper than faulting in all the pages individually at the client. Therefore, in a traditional approach, if a client cache is empty before a query is processed, the cache will also be empty after the query has been processed.

3. IMPLEMENTING CACHE INVESTMENT

Given the background of the preceding section, we can now describe our approach to implementing cache investment. We first show how cache investment can be integrated into the query optimizer and describe basic mechanisms for identifying data items as candidates for cache investment. We then present policies for determining when and for which tables and indices to invest. These policies are invoked each time a query is submitted for optimization at a client.

3.1 Integrating Cache Investment and 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.

One way to implement cache investment is to code it directly into the logic of the query optimizer. Such an approach, however, can disrupt the optimizer's cost modeling and plan selection abilities, and is also quite difficult to implement. In our case, we already had an existing optimizer that was capable of generating hybrid-shipping plans [Franklin et al. 1996], 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.

Our approach to integrating cache investment with an existing optimizer is illustrated in Figures 1 and 2. The starting point is a "traditional" distributed query optimizer for a hybrid-shipping system (see Section 2.3). For example, the query optimizer of System R* [Lohman 1988] or our optimizer of [Franklin et al. 1996] could be used. Such an optimizer carries out join ordering, access path selection, and operator site selection. It works by enumerating alternative plans, estimating the cost of each plan, and choosing the plan with the lowest estimated cost. The details are typically quite complicated and this is one of the reasons why we would like to avoid changing the internals of the optimizer.

Given a traditional hybrid-shipping optimizer, cache investment can be integrated as shown in Figure 2. When the traditional optimizer requests the current status of the cache, the cache investment policy augments the answer with the tables and indices that *should be* cached. In effect, cache investment lies to the optimizer. It *patches* the cache content information passed to the optimizer, so that the optimizer believes that all the data items that should be cached are already present in the cache. This misinformation leads the optimizer to consider placing scan operators for such tables and indices at the client, even if none of their pages are cached, so that the relevant data items are brought to the client and can be subsequently cached.

It is important to reiterate that in this scheme, neither the optimizer's search strategy nor its cost model are 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 cache investment. For example, this approach would not force a scan of a table to be executed at a client if queries using that table could always be executed most efficiently at the server. Such a situation can arise, for instance, if the server is more powerful than the client.

3.2 A Framework for Identifying Caching Candidates

As described in the previous section, the role of a cache investment policy is to determine which data items (tables, partitions of tables, or indices) *should be* cached at the client. We refer to such data items as *candidates*. During query optimization, it is these candidate items for which the cache investment policy will patch the cache content information provided to the optimizer. The decision of whether or not a data item should be considered a candidate is a tradeoff between the cost of initiating the caching of that item (i.e., the *investment cost*) and the expected gain to be realized by caching the item (i.e., the ROI, or *return on investment*).

The investment cost for caching a data item is paid by one or more queries (i.e., the queries that fault in pages of the item). This cost is the difference in response time (or total cost) of plans that initiate the caching compared to the response time (or cost) of the best plans for those queries. Note that if all the pages of a data item are already cached, the investment cost for that item is 0. The ROI, or benefit of caching can be calculated as the cumulative savings in the response time (or cost) for *future* queries that can be achieved while the data item remains cached. Thus, in contrast to investment cost, the computation of ROI ideally requires knowledge of the queries and updates that will occur in the future. Note that if a data item is not used at all in the future, its ROI is 0.

In general, a cache investment policy should consider an item to be a candidate if it meets both of the following criteria:

(1) The ROI is higher than the investment.

(2) The ROI minus the investment is higher than the ROI of the currently cached data item(s) it would replace if parts of it were brought in.

The first criterion ensures that investing in the data item would produce a net gain. The second criterion ensures that only the most valuable data items are kept in the client's cache. Intuitively, an ideal cache investment policy would choose candidates based on perfect knowledge of both the investment and ROI for all data items in the database. Of course, any practical cache investment policy must estimate these quantities. The estimation of ROI is further complicated by the fact that it depends on predictions of future accesses. As with normal query optimization, we rely upon the fact that absolute accuracy in estimations is not necessary for producing reasonable plans and avoiding disastrous ones. In fact, an advantage of our approach to cache investment is that it allows the cost estimation capabilities of the existing query optimizer to be exploited, which can reduce the amount of mechanism that must be added to support estimation techniques.

Recall that cache investment considers indices as potential caching candidates in addition to tables (or partitions of tables). It is important to note that caching decisions are made at the granularity of *data items* such as tables or indices, whereas caching itself is more fine-grained and carried out at the granularity of pages. Due to the nature of table scans, which always read all pages of a table (or partition of a table) and index scans, which selectively read a couple of pages of an index and normally also a couple of pages of the underlying table, there is a subtle difference between what it means for a table or an index to be considered a candidate. When a table is a candidate, that means that *all pages* of the table should be cached at that client. In contrast, for an index, being a candidate means that all pages of the index *and* the underlying base table that are *used in queries* at a client should be cached at that client. We will come back to this point in Section 3.4.

3.3 Cache Investment Policies for Base Tables

The basic *mechanisms* for introducing cache investment into a hybrid-shipping system have been described above. Of course, in addition to mechanisms, *policies* for choosing investment candidates are also needed. We have developed two polices for dynamically choosing investment candidates: *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 data items, while the Reference-Counting policy is simpler; it ranks items by their frequency of use, without explicitly calculating expected ROI's, and ignores investment costs. In this section, we describe the two policies as they are used for tables and partitions of tables in the absence of indices. In the subsequent sections we extend the policies to consider indices.

Of course, heuristics based on frequency of access and various types of cost/benefit analyses have been used in many other memory management and data placement scenarios. The main goal of cache investment is similar to that of any caching approach, namely, to place copies of data closest to where they will most likely be accessed. Thus, it is not surprising that the basic policies for cache investment rely on insights similar to those used in other schemes. The key aspects of cache investment that differentiate it from these other approaches are: 1) its integration with and exploitation of the query optimizer; 2) its applicability to complex relational queries; and 3) its consideration of both base data and index data as investment candidates.

3.3.1 *Maintaining History Information.* Both Reference-Counting and Profitable maintain history information about tables for all client sites. In particular, the information kept for a table at a client is a number that represents a *value* for caching that table at the client. This history information is maintained for all tables that were used in queries at a client. It is kept independently from the (LRU) statistics kept by the cache manager for replacement decisions. The way that values are assigned differs according to the particular policy being used. For both policies, the values of tables 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 [Effelsberg and Härder 1984]. The value of every table is initially set to 0. As described in Equation 1, the value $V_t^c(q)$ of a table t at client c after the execution of query q is multiplied by an aging factor α $(0 \leq \alpha \leq 1)$, and increased by a component $v_t^c(q)$ $(0 \leq v_t^c(q) \leq \infty)$.

$$V_t^c(q) = v_t^c(q) + \alpha * V_t^c(q-1) \tag{1}$$

For both history-based policies, $v_t^c(q)$ is set to 0 if table t was *not* used in query q, 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 table 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 table values can be restricted to only those tables whose value is above a certain threshold. When the value of a table drops below this threshold, the value is set to 0 and the table is ignored until it is again used in a query. Throughout this study, we use a threshold of 0.01 for this purpose.

An important feature of the history-based cache investment policies is that they can easily be extended to deal with updates. If a table is frequently updated, then caching that table (or parts of it) can become less attractive. This fact can be nicely modeled with cache investment by reducing the value of a table with an update, regardless of how updates are implemented in the system and regardless of whether the Profitable or Reference-Counting policy is used. When an invalidation-based cache consistency protocol is used (such as callback locking in SHORE [Carey et al. 1994; Franklin 1996]), then the value of a table is reduced proportionally as a result of updates. More precisely, before a page is modified, the server sends a message to all clients that have a copy of that page in their cache. As a result, all those clients will mark that copy as invalid or give up that copy, as soon as all running transactions that read that page have committed. What is important for cache investment is that a client loses a page of a cached table due to an update, and thus, the value of the table must be reduced proportionally: specifically, if a pages of table t are cached at client c after the execution of query q, and u pages of that table are called back before query $q+1$ is issued, then the value of the table is adjusted as follows:

$$V_t^c(q) := \frac{a-u}{a} * V_t^c(q)$$

When a propagation-based cache consistency protocol is used (such as in SAP R/3 [Kemper et al. 1998]), the value of a table is reduced by the cost of every update message from the server. That is

$$V_t^c(q) := V_t^c(q) - m$$

if an update message with cost m is sent to the client (m depends, among others, on the size of the message). Note that in a system that uses a propagation-based approach, it is possible that the value of a table falls below 0. In this case, cache investment explicitly advises the cache manager to drop its copy of the table. In systems that employ an invalidation-based protocol, such explicit dropping of cached data is not necessary: if, for example, all pages of a table are updated, the value of the table will be reduced to 0 and, at the same time, all the pages will have been invalidated or removed from the cache.

Given the above description on periodic aging and the adjustments made to deal with updates, two questions remain to be answered to instantiate a history-based policy for tables:

(1) How is $v_t^c(q)$ computed?

(2) When is a table 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 for Base Tables.* Reference-Counting is an application of ideas developed for reference-based replacement policies [Effelsberg and Härder 1984] to cache investment. For Reference-Counting, the component $v_t^c(q)$ of Equation 1 is set to 1 if any part of table t is used in query q of client c (i.e., if executing query q involves a table scan of table t), and $v_t^c(q)$ is set to 0 otherwise. Thus, the value of a table (i.e., $V_t^c(q)$) for Reference-Counting is a count of the number of queries in which a table is used, possibly weighted by the recency of those accesses as determined by the α parameter. As a result, the Reference-Counting policy will influence the query optimizer to initiate the caching of frequently used tables.

Unlike the ideal policy described in Section 3.2, Reference-Counting does not compute estimated ROI's for the tables and it ignores the cost of investment. Instead, the Reference-Counting policy tries to maximize the value of the tables stored in a client's cache. This is, essentially, a knapsack problem, and we use the obvious heuristics of packing those tables with the highest value/size ratio [Horowitz and Sahni 1976]. (The same heuristics have also been used for data placement in Bubba [Copeland et al. 1988], WATCHMAN [Scheuermann et al. 1996], and in [Stonebraker et al. 1990]).

This technique is demonstrated by the example shown in Table 1. In the example, the tables 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 tables A and B (i.e., a total value of 800 in this case), and only these two tables would be considered

table	*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

candidates by the Reference-Counting policy. 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.[1] Thus, A, B, and C would be potential candidates.

3.3.3 *The Profitable Policy for Base Tables.* The second history-based policy we study, called Profitable, attempts to more directly estimate the investment cost and ROI for the tables used in a query. The cost for investing in a table t for query q is computed as the difference in cost of the best execution plan for q and the cost of a (potentially suboptimal) plan for q that brings the pages of table t to the client. Note that this cost is zero if the best plan already involves shipping t to the client or if t is already cached at the client. The ROI of a table is taken to be its total *value* at the time of the optimization of query q (i.e., $V_t^c(q)$). To compute this value, $v_t^c(q)$ (from Equation (1), the contribution of query q to the value of t) is computed as the cost of query q without t cached minus the cost of query q *with* t cached. In other words, it is the performance benefit gained for query q by having t cached at the client.

Given these estimates of investment cost and ROI, the Profitable policy considers t to be a candidate for query q if the following three criteria are met:

(1) t is accessed in q.
(2) The value (i.e., the estimated ROI) of t is greater than the investment cost for t in query q and also is greater than the history retention threshold (which as described in Section 3.3.1 is set to 0.01 throughout this work).
(3) The value of t is high enough that t would be fully or partially kept in a cache with a total maximum value (as defined for the Reference-Counting policy above).

When a query is submitted for optimization and execution, the Profitable policy estimates the investment cost and performance benefits gained by caching for each table accessed in the query by performing a series of "what-if" analyses using the query optimizer. This process is similar to optimizer-based approaches for automating physical database design (e.g., [Schkolnick et al. 1988; Chaudhuri and Narasayya 1997]).

In order to compute investment cost, the optimizer is first called to generate the best possible plan for query q given the actual current state of the cache. Depending on the state of the cache, this best plan may have none, some, or all of the *scan* operators placed at the client. Then, for every table t involved in the query that is not already fully cached, the optimizer is called to generate a "t caching" execution

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

plan for the query. To generate a t caching plan, the optimizer is told the actual current state of the cache augmented with the (false) information that table t is also cached. The plan generated in this way is likely to have a *scan* operator for table t at the client, which would enable caching of table t. Using the optimizer's cost model, the Profitable policy re-evaluates this caching plan using the *correct* state of the client's cache. The investment cost for table t is then computed as the difference between this cost and the cost of the best plan for query q.

$v_t^c(q)$, the performance benefit for query q having t cached, is estimated in a similar manner. First, the Profitable policy calls the optimizer to generate a query plan under the assumption that all of the tables used in the query are fully cached at the client. This plan is likely to have all *scan* operators placed at the client because the optimizer believes that all tables can cheaply be read at the client. Then, for every table t involved in query q that is not already fully cached, the optimizer is called to generate a "t comparison" execution plan for the query. To generate the t comparison plan, the optimizer is told that the whole database *except* table t is cached at the client. The resulting plan is likely to have a *scan* operator for table t at the server. $v_t^c(q)$ is then computed as the difference in cost between the fully cached plan and the t comparison plan.

To demonstrate how the Profitable policy estimates investment cost and performance benefits, let us go back to Example 1 of the introduction and assume (for simplicity) that the optimizer's cost model only considers communication costs. Recall that the example involved a join of two relations (A and B), of 1 MB each, stored at two separate servers and produced a result of 0.9 MB. Assume that the client starts with an empty cache. The investment to initiate the caching of table A in the first execution of the query $A \bowtie B$ is computed as follows: (1) Generate the best plan for the query; this plan performs the join at one of the servers and has communication costs of 1.9 MB as described in the introduction. (2) Generate a plan making the optimizer believe that table A is already cached and then re-evaluate that plan. This plan carries out the join at the client and has communication costs of 2 MB. Thus, the investment for A is 2 MB - 1.9 MB = 0.1 MB. (3) Repeat step (2) for table B, which in this case, also results in an investment cost of 0.1 MB.

To estimate the performance benefits of caching tables A and B for the first execution of the query $A \bowtie B$, the Profitable policy would proceed as follows: (1) Generate a plan making the optimizer believe that the whole database is cached; the cost of this plan is 0 MB. (2) Generate a plan making the optimizer believe that the whole database except table A is cached; this plan carries out the join at the client and has a cost of 1 MB (i.e., the cost to ship table A to the client). So, $v_A^c(1) = 1$ MB $-$ 0 MB $=$ 1 MB. (3) Repeat step (2) for table B, which in this case, results in $v_B^c(1) = 1$ MB. These numbers are then plugged into Equation (1) to calculate the total value (i.e., estimated ROI) for each table. The resulting investment costs and ROI estimates are used along with knowledge of the current cache contents to determine if either or both of the the tables should be considered candidates, as was specified at the beginning of this section.

It should be noted that this approach to estimation is less accurate than it could be because the estimation is performed for each table individually rather than for combinations of tables. The investment to bring tables A and B to the client, for example, is estimated to be 0.1 MB for each table or 0.2 MB in total, although it really is only 0.1 MB in total. We chose to perform the estimation for each table separately in order to reduce the number of required "what-if" analyses (i.e., linear rather than exponential in the number of tables in a query). Another source of imprecision in the ROI calculation is that the size of the cache is not considered when the ROI of a table is computed. If the size of the cache is less than 1MB in Example 1, then the estimated ROI of 0.9MB can never be achieved for any table because none of the tables can be fully cached. The situation can get worse for multi-way join queries in which the cache is large enough to hold some tables but not all. We expect, however, that more accurate modeling would not likely result in noticeably better decision making for cache investment, as cache investment decisions are intended to provide benefits over the long-term rather than for any single query execution.

3.4 Cache Investment for Indices

In the previous section, we showed how cache investment decides which tables (or partitions of tables) should be cached at a client (i.e., chosen as candidates) in the absence of indices. In this section, we extend our model to account for the presence of indices. Basically, an index can be seen as a type of *data item* so the cache investment policies devised in the previous subsections remain applicable. There are, however, three important points that must be kept in mind:

—An *index scan* reads only a small fraction of an index.

—It is usually not advantageous to only cache index pages; the referenced pages of the base tables should also be cached.

—Indices and the corresponding base tables should be considered as separate data items, and they should *compete* for space in a client's cache.

The first point makes it necessary to find ways to predict how much cache space would be required to cache an index because the *required space* to cache a data item is an important factor for cache investment with a Reference-Counting or Profitable policy. The required space to cache an index depends on the selectivity of the predicates used in the queries, and we show how to compute this factor in Section 3.4.1. The second and third points make it necessary to adjust our notion of *candidate* data items. If a whole base table is a candidate, then all the pages of that table should be cached; caching all the pages of a table makes it possible to carry out *table scans* at the client efficiently. If an index is a candidate, then all the pages of the index *and* of the corresponding base table that are *used* in queries at the client should be cached; caching all these pages makes it possible to carry out *index scans* at the client efficiently. We describe this impact on the Profitable and Reference-Counting policies in Sections 3.4.2 and 3.4.3. Finally, Section 3.4.4 describes how the presence of indices also affects the way cache investment accounts for updates.

3.4.1 *Space Requirements of Cached Indices.* To estimate the *required space* to cache an index, we first present a very simple model to estimate the number of index and base table pages that need to be accessed to execute one query using the index. For simplicity, we assume that the index is a B^+-tree and we only count the used leaf pages of the B$^+$-tree. Assume that the index is used to evaluate a predicate p of the query, the optimizer's statistics (e.g., histograms) indicate that k tuples satisfy p, the B$^+$-tree has m leaf pages, and the table has N tuples in total, then

$$\frac{k}{N} * m$$

leaf pages of the B$^+$-tree are accessed to evaluate p using the index. Correspondingly,

$$\frac{k}{N} * n$$

pages of the base table need to be accessed to evaluate p, if the B$^+$-tree is a clustered index and n is the total number of pages in the base table. If the B$^+$-tree is an unclustered index, then the number of base table pages that need to be accessed can be estimated using Yao's formula [Yao 1977].

Now, let us consider a situation in which we have estimated that all past queries (i.e., queries 1, 2, ..., $q-1$) of client c could have been executed with $S_i^c(q-1)$ pages of index i and $S_t^c(q-1)$ pages of the corresponding base table t. If the next query q of client c can be executed with $s_i^c(q)$ index and $s_t^c(q)$ base table pages, then all past queries plus this next query can be executed using

$$S_i^c(q) = S_i^c(q-1) + s_i^c(q) - \frac{S_i^c(q-1) * s_i^c(q)}{m}$$

index pages and

$$S_t^c(q) = S_t^c(q-1) + s_t^c(q) - \frac{S_t^c(q-1) * s_t^c(q)}{n}$$

base table pages. In this way, we can predict the space requirements of an index for a stream of queries. The last two equations are derived assuming that queries use index and base table pages according to a uniform distribution. In the presence of skew, however, the overlap would be greater and the required space would be smaller so that these equations can be seen as conservative estimates that typically overestimate the actual space requirements.

It should be noted that some queries can be executed using an index, without accessing the base table at all; an example is a query that asks for the average salary of the top 10,000 `Employees`, which can be executed using an index on `Emp.salary`. In this case $s_t^c(q) = 0$ and the formulae above remain applicable.

3.4.2 *Profitable Policy.* We now turn to extending the Profitable policy to deal with indices. Recall that the Profitable policy identifies tables as candidates based on "what-if" analyses. In the presence of indices, these "what-if" analyses must also be applied to every index *applicable* to a query. An index is *applicable* if one or more of the predicates of the query can be processed using the index (for multi-column indices, at least one predicate involving the first column of the multi-column index must exist in the query), if the index can be used in joins or group-bys, or if the query has an `ORDER BY` clause on the indexed attribute(s).

The estimation of $v_t^c(q)$, the benefit of caching the whole table t, is adjusted as follows. As in the base table case, two plans are considered. The first plan is generated making the optimizer's cost model believe that the whole database including the table, but excluding all the indices on that table, are cached; this plan is likely to have a *table scan* operator at the client because the optimizer's cost model believes that the table can be cheaply read from the client's main memory, but no indices are available at the client. Thus, this Plan 1 is exactly the plan that would use the cached copy of the whole table, if the whole table were cached. Plan 2 is generated telling the optimizer that the whole database, excluding that table and excluding all the indices on that table, is cached; this plan is likely to carry out *scan* operators on that table at a server (with or without indices). The benefit of caching the whole table (i.e., $v_t^c(q)$) is then computed as the difference in cost between Plan 2 and Plan 1.

For an index i, the following two plans are generated: Plan 1 is generated telling the optimizer that the whole database including that index and the entire corresponding table, but excluding all other indices on that table, are cached. This plan is likely to have an *index scan* operator at the client. Plan 2 is generated telling the optimizer that the whole database excluding that index, the corresponding table, and all other indices on that table are cached; this plan is likely to carry out all *scan* operations at a server (with or without indices). The benefit of caching index i (i.e., $c_i^c(q)$) is then computed as the difference in cost between the second and the first plan.

The investment to initiate the caching of a table is carried out in exactly the same way as proposed in Section 3.3.3. The investment to initiate the caching of an index is computed in an analogous way.

3.4.3 *Reference Counting Policy.* The Reference-Counting policy, as before, uses a simplified approach to determine which indices and tables should be cached at a client. When a query is submitted at a client, the Reference-Counting policy first asks the optimizer to generate the best possible plan for this query, given the actual state of the client's cache and also telling the optimizer that indices and tables that have a high *value/size* ratio from past queries should be cached. The plan returned by the optimizer is then executed to produce the query results, thereby possibly bringing data into the client's cache.

The plan is also analyzed by the Reference-Counting policy in order to update the *values* of the indices and tables involved in the query plan. If, for example, the plan generated by the optimizer involves an *index scan*, then the value of the used index is incremented whereas the values of the whole base table and all the other indices on that table are not incremented. If, on the other hand, the plan involves a *table scan*, then the value of the base table is incremented whereas none of the values of the indices are incremented. This way, as we will see, the Reference-Counting policy is able to make the right decisions in many situations, but it cannot always be perfect because the Reference-Counting policy finds out which indices and tables are good to cache, but it does not really know how much better it would be to cache say, one index rather than another index or a whole table (see Section 4.5). In any case, the Reference-Counting policy has almost no additional overhead because it calls the optimizer only once for every query—just like a "traditional" system.

3.4.4 *Dealing with Updates.* As described in Section 3.3.1, the values of data items must be adjusted in the presence of updates. Here, we concentrate on the case where a callback locking (i.e., invalidation-based) approach is used to maintain the consistency of cached pages of base tables as well as of cached leaf pages of indices as proposed in [Zaharioudakis and Carey 1997] for SHORE. Again, the idea is to reduce the value of an index proportionally in the presence of updates. We must keep in mind, however, that we consider index i to be worth $V_i^c(q)$ only if all $S_i^c(q)$ relevant index pages and all $S_t^c(q)$ relevant base table pages are in the cache. So, we must reduce the value of an index when both index and base table pages are invalidated. After the invalidation of u index and base table pages, the value of index i is, therefore, adjusted as follows:

$$V_i^c(q) := \frac{S_i^c(q) + S_t^c(q) - u}{S_i^c(q) + S_t^c(q)} * V_i^c(q)$$

Using this formula, we are being conservative in two ways. One, we assume that index and base table pages are invalidated independently. This is a reasonable assumption for unclustered indices, but it is clearly not true for clustered indices: for clustered indices, an update that invalidates an index page always also invalidates one of the referenced cached base table pages. For clustered indices, we therefore only take invalidations of base table pages into account and use the following formula to adjust the value of an index after u_t base table pages have been invalidated:

$$V_i^c(q) := \frac{S_t^c(q) - u_t}{S_t^c(q)} * V_i^c(q)$$

Another conservative assumption that we make in both formulae above is that we assume that every time a base table page is invalidated, the page is actually referenced by one of the cached pages of the index. We could reduce the imprecision of this assumption by keeping track of which base table pages are referenced by which cached index pages, but we have not implemented this yet.

Furthermore, as described in Section 3.3.1, it would also be possible to model the effects of a propagation-based protocol to maintain the consistency of cached indices. We, however, do not know of any system that uses such a propagation-based protocol for cached indices, so we omit the discussion for brevity.

3.5 Summary

We now briefly summarize our cache investment policies. All policies generate a set of *candidate* data items (tables, partitions of tables, and indices) that *should* be cached at a client and attempt to initiate the caching of (the relevant parts of) candidate data items by patching the cache content information provided to the query optimizer. We introduced two history-based policies, Reference-Counting and Profitable, in which the choice of candidates depends on the sizes of the (relevant parts of) the data items, the size and contents of the cache, the past history of queries submitted at a client, and on the presence of update operations initiated at other clients. The use of history is intended to enable these policies to adapt to a client's workload. The Reference-Counting policy considers only the most frequently used data items to be candidates and ignores the cost of investment. The Profitable policy calculates an expected ROI for each data item and chooses as candidates those data items who have the highest expected ROI and whose investment cost is less than their expected ROI. Both policies have extensions to deal with indices, but the basic mechanisms are the same for both the indexed and non-indexed cases.

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. PERFORMANCE EXPERIMENTS AND RESULTS: EXPERIMENTS WITH SHORE

In order to study the effectiveness of the cache investment policies in a real client-server database system we implemented them as described in the previous section and integrated them with a query optimizer for the SHORE distributed database system. We also performed a simulation-based study in order to explore certain tradeoffs of cache investment that we could not study using SHORE (i.e., the performance of cache investment under various different client/server configurations). In this section, we describe the performance experiments and results we obtained using SHORE. We describe our simulation results in the next section.

4.1 Baseline "Static" Policies

In addition to the history-based policies for choosing investment candidates described in the previous section, we also implemented two "static" policies that are used as baselines in the study. Static policies assign fixed values for investment and ROI to tables independent of any history. The baseline policies we use are: 1) the *Conservative* policy, which assigns values such that it never considers any tables as candidates, and 2) the *Optimistic* policy, which assigns values such that it always considers all tables as candidates. While these polices may seem simple, they in fact, correspond to the way that existing systems are built.

4.1.0.1 *Conservative Policy* . The Conservative policy assigns the ROI of every item to be 0 and the investment to be ∞ so it never considers any items to be candidates. Thus, the behavior of the Conservative policy corresponds to a hybrid-shipping system without cache investment; query optimization is carried out in the traditional way. As described in Section 3.1, the optimizer places all scans at servers without cache investment because placing scans at a client to initiate caching always comes at an additional cost; as a result, the cache of a client is always empty and caching is not exploited. Going back to the examples of the introduction, the Conservative policy would make the right decision for Example 2 (i.e., execute the query at the server), but it would make the wrong decision for Example 1.

The behavior of the Conservative policy is similar to that of most commercial relational (i.e., query-shipping) database systems which do not employ caching. Keep in mind, however, that hybrid shipping with a Conservative cache investment policy can perform better than query shipping because hybrid shipping provides the flexibility to execute joins (and other operators) at clients if all servers are heavily loaded [Franklin et al. 1996]. With a Conservative policy, furthermore, the choice of

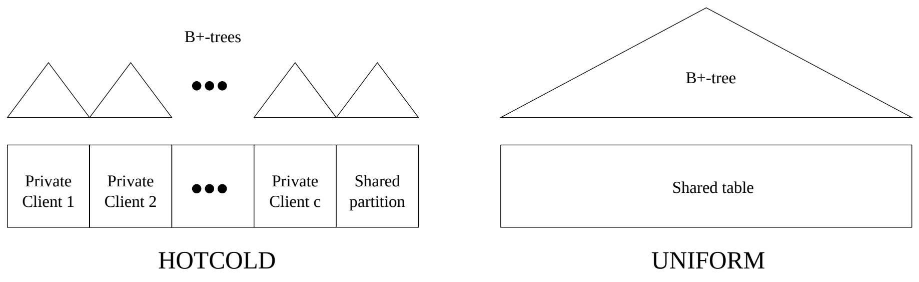

Fig. 3. HOTCOLD and UNIFORM Databases

the right cache replacement policy (e.g., LFU, LRU, or LRU-k) is irrelevant because the cache remains empty.

4.1.0.2 *Optimistic Policy .* The Optimistic policy is so named because it sets the ROI of all items to be ∞, and the investment to be 0. It therefore considers all tables to be candidates and attempts to bring all items accessed by the query into the client's cache if they are not already there. For the examples of the introduction, the Optimistic policy would make the right decision for Example 1 and the wrong decision for Example 2.

The behavior of the Optimistic policy is similar to that of a *data shipping* architecture which places all scans at the client in order to exploit client caching. Note, however, that hybrid shipping with an Optimistic cache investment policy can perform better than data shipping because it would not always place scan operators at the client, even if all the data are cached in the client's main memory (Section 3.1). In addition, hybrid shipping with an Optimistic policy makes it possible to execute scans at a client (exploiting the cache) and ship intermediate results to servers for further query processing in order to exploit the server resources, too [Franklin et al. 1996].

4.2 Benchmark Specification

The workloads used in the experiments with SHORE were derived from the workloads used by Zaharioudakis and Carey in their performance study on the use of indices in client-server database systems [Zaharioudakis and Carey 1997]. Our version of the benchmark is slightly simplified; the main difference is that we had to implement the update transactions differently in our prototype, but we did carry out the same kinds of operations in our experiments as Zaharioudakis and Carey did in theirs. Like Zaharioudakis and Carey, we generated two different databases called the HOTCOLD and the UNIFORM databases, shown in Figure 3. The UNIFORM database consists of one large table with an unclustered B^+-tree index. In the HOTCOLD database, this table is partitioned into $c + 1$ horizontal partitions, where c is the number of clients in the system. In this workload every client has its own *private* partition to which it has exclusive access, and there is one *shared* partition used by all clients. As shown in Figure 3, the HOTCOLD database has one unclustered B^+-tree index for every partition.

Table 2 shows our parameter settings for the two databases. Both databases are about 40 MB (without indices): 200,000 objects and 20 objects per 4 KB page.

	HOTCOLD	UNIFORM
NumObjects	200,000	200,000
ObjectsPerPage	20	20
ObjectsPerPart	40,000	—

Table 2. Database Parameter Settings

The HOTCOLD database is partitioned into equally-sized partitions so that every partition has 40,000 objects in a configuration with four clients.

Corresponding to the two databases, there are two workloads. In both workloads, every client issues a single stream of range queries and update operations; the queries and update operations of all clients run concurrently. In both workloads, we varied the frequency of update operations. Furthermore, in both workloads a range query reads 100 objects using an unclustered index. In the HOTCOLD workload, however, 80% of the range queries of a client involve the client's private partition; the remaining 20% involve the shared partition. In the UNIFORM workload, naturally, all queries involve the single (shared) table.

As stated above, we implemented the update operations differently than Zaharioudakis and Carey: in our version of the benchmark, an update operation modifies one object chosen randomly from the whole database (i.e., disregarding private and shared regions). Each update involves one insert and one delete operation on one of the indices. In the original Zaharioudakis and Carey benchmark, update operations navigate through the database and modify objects on their way. In the HOTCOLD workload of the original benchmark of [Zaharioudakis and Carey 1997], 80% of the update operations involve a client's private partition; the remaining 20% go anywhere in the database (including other clients' private partitions). Thus, our update operations are more light-weight than theirs and our update operations are less skewed than theirs in the HOTCOLD workload. In essence, however, both kinds of update operations do essentially the same thing: they modify objects and indices.

We also studied a third workload that we call the POINTRANGE workload, which was not described in [Zaharioudakis and Carey 1997]. This workload uses the HOTCOLD database and involves point and range queries; here, point queries read one shared object using an index, and range queries read 100 private objects using an index, as before. As an example of this kind of workload, consider a company with sales representatives that ask for details of specific parts (point queries on the "shared" part table) and for sales and customer information of their specific region (range queries on "private" sales and customer partitions). As in the HOTCOLD and UNIFORM workloads, we vary the frequency of update operations from 0% to 80%. The settings of the most important parameters of all three workloads are summarized in Table 3.

4.3 Software and Hardware Used

We used Version 1.0 of SHORE, enhanced with code for client-side index caching which was described in [Zaharioudakis and Carey 1997] and which is not part

	HOTCOLD	POINTRANGE	UNIFORM
FreqSharedQueries	80%	50%	100%
FreqPrivQueries	20%	50%	—
KindSharedQueries	range	point	range
KindPrivQueries	range	range	range
FreqUpdates	0%–80%	0%–80%	0%–80%
Database	HOTCOLD	HOTCOLD	UNIFORM

Table 3. Workload Parameter Settings

of the official SHORE release.[2] A detailed description of SHORE can be found in [Carey et al. 1994]. SHORE provides state-of-the art implementations for record management, B^+-trees, concurrency control and recovery, client-side caching, etc. The current version of SHORE, however, lacks support for query processing so we had to integrate our own query optimizer and basic relational query operators for table scans and index scans.

In all our experiments, queries are optimized fully at run-time. The optimizer we used is a randomized query optimizer which is based on the approach described in [Ioannidis and Kang 1990] extended to carry out site selection in addition to other decisions such as join ordering, as described in [Franklin et al. 1996]. In fact, we were able to use the same optimizer employed in that earlier study because cache investment requires no changes to the optimizer (as described in Section 3.1). In all experiments presented in this paper, the optimizer was configured to minimize the *response time* of a query according to the model of [Ganguly et al. 1992]. Our implementation of that model considered CPU costs, network costs, and costs for random and sequential disk I/O. To calibrate the cost model, we ran a separate set of experiments in our experimental environment; in particular, we set the client and server CPU and disk parameters of the cost model separately and we took the load situation of the server into account when we set the server CPU and disk parameters. Furthermore, the cost model uses information about the size and the contents of a client's memory and cache; this information is refreshed and possibly patched (as described in Section 3.1) by a cache investment policy every time a query is optimized.

In a client-server architecture, there is a question of where to optimize a query [Hagmann and Ferrari 1986] and where to keep the statistics for cache investment: at the client or at the server. Since we wanted to change the SHORE server code as little as possible, we decided to run an instance of the optimizer and also to carry out cache investment at every client. Our results, however, would have been the same in a different architecture that, say, carries out query optimization and cache investment at servers. This is because an optimizer at the server would have generated identical plans (all clients had up-to-date statistics) and the overheads of optimization were very small in our experiments with SHORE. (We discuss optimization overheads for complex queries in Section 6.)

In all our experiments with SHORE, the database was stored at a single, ded-

[2]Specifically, we used the *hybrid caching* approach of [Zaharioudakis and Carey 1997]. *Hybrid caching is a particular protocol to maintain the consistency of cached index pages.*

icated server machine and queries and updates originated from four concurrent client machines. The server was a SUN SPARCstation 20/502MP with two 50MHz SPARC processors, 96MB of main memory, and a 4 GB disk drive. The four client machines were SUN SPARCstation 10/20s with a 33MHz SPARC processor, 64MB of main memory, and a 2 GB disk drive each. The size of the buffer pool at the server was set to be 20 MB (i.e., 50% of the size of the database), and the size of the cache of a client machine was set to be 8 MB (i.e., 20% of the size of the database) in all experiments reported here. All client disks were only used to store software and as swap space; i.e., client disks were not used to cache data. The client machines and the server were connected by a 10 Mbit/sec Ethernet, and Solaris 2.6 was the operating system installed on all machines.

4.4 HOTCOLD Workload

Figure 4 shows the average response time of the queries in the HOTCOLD experiments, varying the frequency of update operations.[3] The figure shows clearly why cache investment with a history-based policy is needed to achieve acceptable performance: both the Optimistic and the Conservative policy show poor performance. This poor performance is due to either over-use of the client resources (in the case of the Optimistic policy) or non-use (in the case of the Conservative policy). Using the Conservative policy, for instance, all queries have very high response time because they are executed at the server, the bottleneck of the system. Using the Optimistic policy, the queries involving the private data of a client can be executed very efficiently at the clients because most of a client's private data is cached at the client, but the queries involving shared data are carried out with very high extra cost at the client because the shared data cannot be cached at the client (the cache is too small). Recall that the Conservative and Optimistic policies correspond to traditional ways of constructing client-server database systems: the Optimistic policy mimics a data shipping system and the Conservative policy mimics a query shipping system or a hybrid-shipping system without cache investment.

In contrast, the history-based cache investment policies show significantly better performance using either the Profitable or the Reference-Counting policy here. Both history-based policies outperform the Conservative policy because they advise the optimizer to execute all queries involving private data at the clients, thereby effectively using the clients' caches. Both history-based policies outperform the Optimistic policy because they advise the optimizer to execute all queries involving shared data at the server, thereby avoiding unnecessary overhead to move shared data to the clients. In this experiment, the differences between the Profitable and the Reference-Counting policy are marginal for two reasons: First, the Reference-Counting policy makes good decisions because the frequency of access is indeed an indicator for how useful it is to cache data. Second, the overhead of the "what-if" analyses of the Profitable policy is very small because these queries only involve one table and one index.

Figure 5 shows the cost of the update operations in the HOTCOLD workload. Again, the two history-based policies have the best performance, the Optimistic policy is in the middle, and the Conservative policy is the worst. Although the

[3]For every data point, we ran several hundred queries in order to minimize effects of randomness.

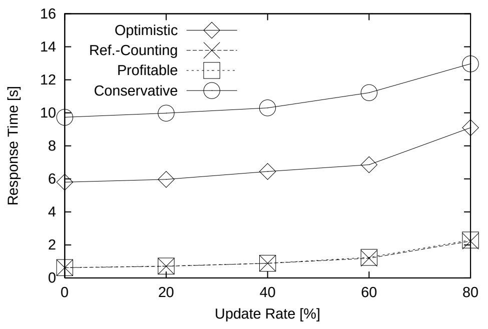

Fig. 4. Resp. Time of Queries (secs)
HOTCOLD Workload

Cost of Updates	
Optimistic	0.55 secs
Conservative	0.64 secs
Ref.-Counting	0.38 secs
Profitable	0.38 secs

Fig. 5. Resp. Time of Update Ops (secs)
HOTCOLD Workload, 20% Update Prob.

Conservative policy performs no work for cache consistency (since it does not use client caches), it still suffers because of the high load that is imposed on the server. In contrast, the other approaches are able to spread much of the load among the clients, so that they outperform the Conservative policy in this workload, even for update operations.

4.5 POINTRANGE Workload

Figure 6 shows the average response time of the queries in the POINTRANGE workload. The best strategy for this workload is to execute all range queries at the client (i.e., cache the private data) and to execute all point queries on shared data at the server. This is because the benefit of caching data that is used in range queries is much higher than the benefit of caching data that is used in point queries. Point queries are cheap no matter where they are executed. In contrast, range queries are cheap only if all the relevant index and base table pages can be found in the client's cache; otherwise, range queries incur significant random disk I/O costs at the server.

As can be seen in Figure 6, the Profitable policy is able to make the right decisions for this workload and shows the overall best performance. The Reference-Counting policy, on the other hand, cannot tell whether it is better to cache private or shared data: both are used in queries with the same frequency (50%) so that the Reference-Counting policy sometimes executes "shared" point queries at clients and "private" range queries at the server and vice versa depending on the current (random) state of its counters at the clients. The Optimistic policy forces the optimizer to execute all queries at the clients and shows quite good performance in this workload because it achieves reasonably high hit rates in the client cache for the range queries on the private data. The Optimistic policy still performs worse than the Profitable policy; when there are no updates, the Optimistic policy is outperformed by the Profitable policy by a factor of two. As in the previous experiment, the Conservative policy shows the overall worst performance. In this case, this is because it carries out all queries at the server and, therefore, always pays a very high price to execute the range queries.

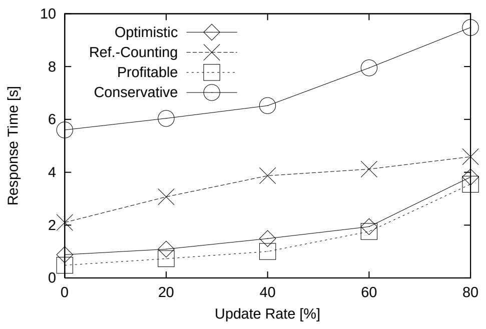

Fig. 6. Resp. Time of Queries (secs)
POINTRANGE Workload

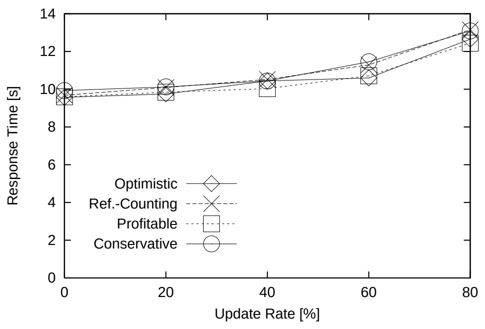

Fig. 7. Resp. Time of Queries (secs)
UNIFORM Workload

4.6 UNIFORM Workload

Figure 7 shows the average response time of the queries in the UNIFORM workload. In this particular configuration (four clients, one server, 20% cache at every client), all four approaches show essentially the same performance. This is because the client caches are just large enough so that the gains achieved by caching are the same as the additional overhead to fault in index and base table pages in the case of misses. With larger client caches, the Optimistic policy would outperform the Conservative policy, and the two history-based policies would behave like the Optimistic policy. With smaller client caches, the Conservative policy would outperform the Optimistic policy and the two history-based policies would behave like the Conservative policy. In the next section we more closely examine how the client/server configuration impacts cache investment using a simulation model.

5. IMPACT OF DIFFERENT CLIENT/SERVER CONFIGURATIONS

The previous section showed the performance tradeoffs of the four cache investment policies for three different workloads and one specific client/server configuration (i.e., 4 clients with 8 MB of cache each and a server with 20 MB of cache). In a nutshell, those experiments showed that the kind of query (i.e., the selectivity of query predicates) and the skew in which data is accessed (i.e., access probabilities) significantly affect the performance of the four cache investment policies: the higher the skew, for example, the better the Reference-Counting policy performs and the worse the Conservative policy performs. In this section we more closely examine the impact of client/server configuration issues on the performance of the various policies.

5.1 Simulation Environment

For this set of experiments, we extended the hybrid-shipping simulator used in previous work on client-server query processing [Franklin et al. 1996]. The simulator accurately models the resources of a distributed database system (e.g., CPUs, network, disks) and the operations of a distributed query engine (e.g., scans and hybrid-hash joins). Furthermore, the simulator models an LRU buffer manager for the client and server caches. The details of this simulation model are described in [Franklin et al. 1996] and are beyond the scope of this paper. Here, we just want to note that we set most of the parameters of the simulator to model fairly common hardware. That is, we set the network bandwidth to 100 Mbit/sec (i.e., an Ethernet), the CPU speed of the clients and the servers to 50 Mips (i.e., SUN SPARC 20 workstations), and the disk parameters for seek, latency and data transfer to model disk drives with an average cost of roughly 3.5 msecs per page for sequential I/O and 11.8 msecs per page for random I/O. In the experiments, we vary the size of the client and server caches and the number of clients and servers. The most important simulation model parameters and their settings are described in Table 4.

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 per site
ClMemory	2 - 40	client's main memory (% 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 4. Simulator Model Parameters and Default Settings

In the simulation, the queries are optimized using the same optimizer as in the experiments with SHORE. Again, the optimizer was configured to minimize the response time of a query using the model of [Ganguly et al. 1992], and the cost model took CPU, disk I/O, and network costs as well as the sizes and contents of the clients' and servers' memories into account. We did, however, calibrate the cost model separately for each client/server configuration studied; for example, we adjusted the servers' CPU and I/O bandwidth to the number of concurrent clients.

The database used in these experiments 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 and not replicated. In experiments with 10 servers, each server stores exactly 10 relations. The database has no indices because indices are not useful for the kind of workload we used in this simulation study. As stated above, the main focus of this simulation study is to study the impact of different client/server configurations on the performance of a cache investment policy, and therefore, we used a very simple workload: every client submits

[4]As with SHORE, the database and relation sizes are kept small in order to achieve acceptable experiment running times.

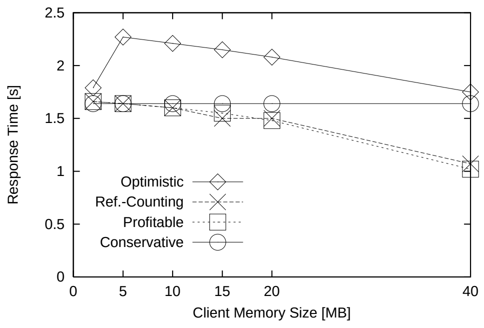

Fig. 8. Response Time, 1 Client, 1 Server
Uniform Join Workload, Vary Client Memory

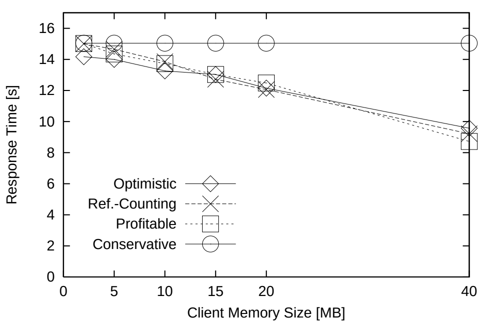

Fig. 9. Response Time, 10 Clients, 1 Server
Uniform Join Workload, Vary Client Memory

a stream of simple two-way functional join queries. In every query, each relation is used with the same probability (i.e., a uniform distribution), and the result of every join query has 10,000 tuples of 100 bytes (1MB), just like the base relations. For all queries, the optimizer chose to use hybrid-hash joins.

5.2 Single Server Environment

In all the experiments with SHORE the Optimistic policy showed as good or better performance than the Conservative policy. Also, all four cache investment policies showed the same performance in the experiments for a *uniform* workload. Figures 8 and 9 show that both of these observations are not necessarily true if we consider different client/server configurations. Figure 8 shows that if the server is lightly loaded (only 1 client is active) and the cache of the client is small, the Conservative policy significantly outperforms the Optimistic policy: in this case, the advantages of caching are lower because the server is only lightly loaded and the additional costs of the Optimistic policy for shipping data to the client are higher because the cache is small. As the size of the client's cache increases, the performance of the Optimistic policy improves as less data must be shipped to the client, but even with a relatively large cache of 40MB, the Optimistic policy is outperformed by the Conservative policy in this scenario.

Figure 8 shows that, again, the history-based policies show the best performance in this case. Here, the history-based policies fill the client's cache with (random) data and process all queries that involve that data at the client. Queries that use other data are processed at the server. As a result, the performance of the history-based policies improves with a growing cache (more data can be kept in the cache), just as the performance of the Optimistic policy, but unlike the Optimistic policy, the history-based policies do not pay the price of unnecessarily moving data to the client.

Figure 9 shows that the situation is quite different if the server is heavily loaded (10 concurrent clients). In this case, the Conservative policy shows the worst performance even for fairly small client cache sizes because the Conservative policy does not take advantage of caching at all. The other three policies show the same performance although they actually behave quite differently: the Optimistic policy, again, carries out all queries at the clients whereas the history-based policies act as they did in the previous case, running queries on cached data at the client, and others at the server. In situations in which the server is heavily loaded, this *selective* behavior of the history-based policies, however, does not provide any performance advantages.

Figure 10 assesses the performance of the four policies in situations in which communication costs dominate the query response times. (Communication costs did not play a dominant role in any of the other experiments in this section.) For this experiment, we set the network bandwidth to 1Mbit/sec in the optimizer's cost model in order to "simulate" a wide-area network. To focus on network usage, Figure 10 plots the average number of (4KB) pages sent from the server to the client per query. We see again (as in Figure 8) that the Optimistic policy is significantly outperformed by the Conservative policy; by as much as a factor of 2 in the extreme case of this experiment. If we increase the selectivity of the join predicate (i.e., reduce the size of the query result), the difference in communication costs between the Optimistic and Conservative policies becomes even larger. While the Optimistic policy causes all base data used in a query to be shipped from the server to the client, the Conservative policy tends to result in plans in which only the (smaller) query result is shipped from the server to the client As seen in Figure 10, the Reference-Counting and Profitable policy are again able to adapt and show as good performance as the Conservative policy for small cache sizes and slightly better performance than the Conservative policy for large cache sizes.

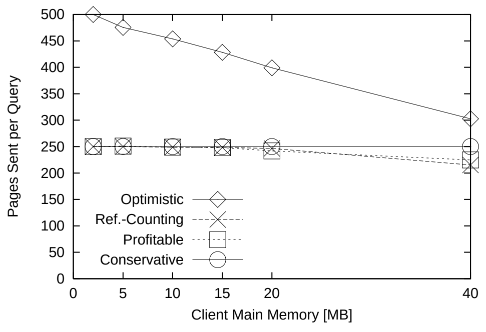

Fig. 10. Comm. Cost, 1 Client, 1 Server Uniform Join Workload, Vary Client Memory

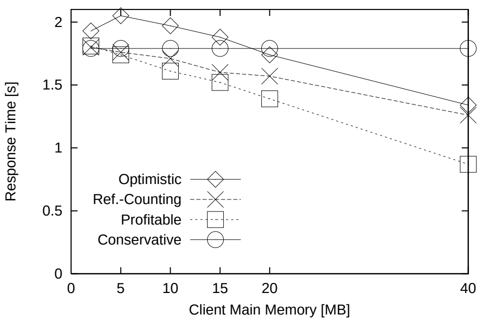

Fig. 11. Resp. Time, 1 Cl., 10 Hetero. Servers Uniform Join Workload, Vary Client Memory

5.3 Multiple, Heterogeneous Servers

We now turn to a scenario in which the database is stored on 10 heterogeneous servers. In this experiment, exactly 10 relations reside on every server. 5 servers are *fast*: they have 40MB of main memory each so that all requests to these servers can be handled "in-memory." The other 5 servers are *slow*: they only have 2MB of main memory each so that most requests to these servers involve disk I/O.

The key result of this experiment is that the Reference-Counting policy is, again, not always able to make the right decisions. As shown in Figure 11, the Reference-Counting policy is significantly outperformed by the Profitable policy, just as for the

POINTRANGE workload of Section 4.5. The best strategy here is to try to cache as much data as possible from the *slow* servers in order to avoid interaction with them. The Profitable policy is able to follow that strategy based on its strategy of caching data that promise the highest ROI. The Reference-Counting policy, on the other hand, caches data from the *fast* and *slow* servers alike because all the data is accessed with the same probability. Likewise, the Optimistic policy is not able to selectively cache the data from the *slow* servers and, therefore, shows suboptimal performance. Due to the high cost for moving data to the client, the Optimistic policy shows the overall worst performance for small client cache sizes in this experiment. The Conservative policy, again, shows relatively poor performance for not exploiting the client's cache at all and, thus, for having to continuously interact with the *slow* servers.

6. EXAMINING CACHE INVESTMENT OVERHEADS

6.1 CPU Costs of Query Optimization

The focus of the previous experiments was on the quality of query plans. We now turn to a discussion of the overheads of the four cache investment policies and their impact on query optimization times. 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. Table 5 shows the results for workloads with 2-way and 5-way join queries.

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

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

There are three ways in which a cache investment policy influences the cost of query optimization. A cache investment policy impacts optimization times most significantly if it calls the optimizer several times in order to carry out "what-if" analyses. This effect can be observed in Table 5 by comparing the optimization times for the Profitable policy and the other three policies. In order to perform its estimations, the Profitable policy must generate two plans for every relation and index involved in the query plus a final plan for the query to execute the query. For five-way join queries without any indices, the optimizer must, therefore be called eleven times using the Profitable policy. On the other hand, the optimizer is only called once using any of the other three policies, regardless of the complexity of the query. For exactly this reason, the optimization times are significantly higher for the Profitable policy than for all the other policies. Note, however, that the optimization time for the Profitable policy can be reduced by *using* the same join order for all plans generated during the "what-if" analyses. That is, the Profitable policy can first generate a complete plan for a query, and then re-apply the optimizer's site selection phase in order to generate alternative plans for the "what-if" analyses (i.e., only the optimizer's site selection rules are applied to carry out the "what-if" analyses). Since join ordering is the most expensive part of query optimization, this trick saves a great deal of optimization cost and due to this trick, the optimization time of the Profitable policy is only about a factor of four (rather than eleven) times higher than the optimization time of the other policies for the 5-way join queries.

Another way in which a policy can impact optimization times is by maintaining statistics for a history of queries. Obviously, neither the Optimistic nor the Conservative policy induce such overheads, and the overheads to maintain the history are negligible for the Reference-Counting and Profitable policy, too: for every query, both policies simply need to adjust the *values* of tables and indices used in the query, and this can be done in fractions of seconds no matter how complex the query is.

The third way in which a policy can impact optimization times is by increasing the search space of query optimization, and this effect is the cause for the differences in optimization times between the Optimistic, Conservative, and Reference-Counting policies. Using the Optimistic policy, the search space is the largest because the optimizer is free to place all scan operators at clients and at servers, and thus, the Optimistic policy has the overall highest optimization times of these three policies. Using the Conservative policy, the optimizer only places scan operators at servers, and thus, the Conservative policy has the lowest optimization times. The Reference-Counting (and Profitable) policies are somewhere in between: they allow the optimizer to place scan operators of tables that are or should be cached at the clients or at the servers and to place scan operators of all the other tables only at servers.

6.2 Tuning the History-Based Policies

In all the previous experiments, the aging parameter α for the Reference-Counting and Profitable policies was set to 0.9. Tuning the two 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 influence the performance of a history-based policy if the workload of a client changes. Intuitively, the smaller α, the faster the history-based policies can adjust to changes in the workload at a client, but the more sensitive the history-based policy becomes to transient changes in the workload at a client. To study how significant this effect is, we examined the quality of plans produced by the Reference-Counting and Profitable policies in the presence of workload shifts for different settings of α. The results of these experiments are briefly described in this section.

To perform this study, we introduced the notion of "sessions" into the workload. Each experiment involves a sequence of sessions. In every session, a client issues a stream of two-way functional join queries (two 1MB tables with a 1MB query result as in Section 5), and the relations involved in a query are chosen using a (standard) Zipf distribution (with parameter $\theta = 1$). That is, within a session, certain (*hot*) relations are used more often than other relations. Between sessions, the Zipf distribution is altered in such a way that different relations are *hot* in different sessions. In the experiment, we varied the length of a session (i.e., the number of queries per session) in order to find out whether cache investment is able to adjust to a new session quickly enough.

Figure 12 shows the average scaled cost of the queries, varying α and the length of a session, when using the Reference-Counting policy. A scaled cost of 1 indicates that the Reference-Counting policy produced the best plans at these settings; a scaled cost of, say, 1.3 means that the response time of the plans was, on an average, 30% higher than under the best setting of α. The results for the Profitable policy are not shown because they are almost identical in this particular case.

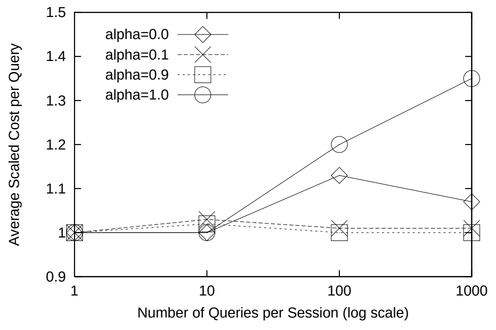

Fig. 12. Average Scaled Cost of Queries, Ref.-Counting, Vary α
Reference-Counting Policy, Mixed Workload, 10% Client Memory

Surprisingly, all four settings of α show fairly much the same performance: there are no real disasters. Setting $\alpha = 1$ shows the overall worst performance. In this extreme case, 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 the *hot* relations of previous sessions retain their high values. In the other extreme, $\alpha = 0$, 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 because the *hottest* relations are often used in two consecutive queries, and as a result, $\alpha = 0$ shows, on an average, better performance 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 or anything else in between. Therefore, these settings show the best performance 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.

7. RELATED WORK

Cache investment and the ideas and results presented in this paper are based on and related to a variety of different concepts used to implement distributed database and information systems. First of all, there has, of course, been a great deal of work in the area of caching for database systems. In [Franklin et al. 1996], 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 [Roussopoulos and Kang 1986; Chen and Roussopoulos 1994; Stonebraker et al. 1990; Keller and Basu 1994; Dar et al. 1996], 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 is, therefore, not directly addressed in this paper. For example, it is still unclear how the consistency of cached query results can be maintained most efficiently in the presence of updates, what the right storage structures for query result caching are, and how a query optimizer can take cached query results into account; before worrying about cache investment, these fundamental issues for query result caching must be addressed.

In general, cache investment is just as important in an architecture that supports query result caching as it is in a more traditional architecture. In this case, any query operator (not just scans) can initiate the caching of data; cache investment is needed to influence the placement of *all* operators and to decide which intermediate results to materialize and cache. Cache investment is more complicated in a query caching architecture by the fact that there is an exponential explosion in the number of "what-if" analyses that must be carried out if, say, the Profitable policy is applied in a naive way. Once query result caching technology is more mature, we intend to study the impact of query result caching on cache investment in detail.

Caching can be considered a special (i.e., dynamic, demand-driven) form of replication, and various dynamic replication algorithms which are related to cache investment have been studied in the literature. Wolfson, Jajodia, and Huang devised the ADR algorithm for replication which establishes a replica of an object at a site if the object is accessed more often at that site than updated at all the other sites [Wolfson et al. 1997]. (Related work by the same authors can be found in [Wolfson and Jajodia 1992b; Wolfson and Jajodia 1992a; Huang et al. 1994].) Applied to our context, the ADR algorithm would decide to establish a copy of a data item at a client if the data item is accessed at that client twice before it is updated by another site, and thus, the Reference-Counting policy bears some similarity with the ADR algorithm. Furthermore, Sidell et al. propose an economic model in order to decide where to keep replicas of tables and indices [Sidell et al. 1996], and thus, the Profitable policy bears some similarity with the replication approach taken in Mariposa.

There are, however, significant differences between caching and replication, and consequently, our approach to cache investment differs significantly from the replication approaches presented in [Wolfson et al. 1997] or [Sidell et al. 1996]. Most importantly, caching is a by-product of query-processing and thus, cache investment is closely integrated with query optimization. In contrast, replication decisions are carried out at server machines independent of the processing of any particular query. As a result, the investment to establish cached copies of data is smaller than the investment to replicate data because caching takes effect when data must be processed and shipped to the client in any case, as part of processing a query. Furthermore, caching is fine-grained (granularity of pages) whereas replication is usually coarse-grained (granularity of relations, horizontal partitions of relations, or indices), and therefore, our models used for cache investment differ significantly from the models proposed for replication. To make this point clearer, consider a scenario in which a client repeatedly asks for information about some specific customer; replicating the whole *Customer* table at the client is unlikely to be beneficial, but caching the right page of the *Customer* table together with the necessary index pages might very well be attractive.

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., [Palmer and Zdonik 1991; Curewitz et al. 1993; Kraiss and Weikum 1998]).

More recently, several researchers have investigated the use of cost and benefit calculations for determining when to keep or replace cached data. WATCHMAN [Scheuermann et al. 1996] 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. One difference between WATCHMAN and cache investment is that WATCHMAN is only effective if the same query is repeatedly asked by different users. Because cache investment involves the caching of base data and indices (rather than query results), the cached data can be used for any kind of query that involves that base data and indices; therefore, cache investment can be applied more broadly than WATCHMAN. A second difference is that updates are ignored in WATCHMAN because WATCHMAN is geared towards a data warehouse environment. As a consequence, the architectures of WATCHMAN and cache investment differ significantly: WATCHMAN can be implemented as an additional layer *on top of* a database system or data warehouse. In contrast, cache investment is *integrated into* the query processor of a (distributed) database system and occasionally influences the optimizer to generate suboptimal plans in order to achieve its somewhat different goals.

Another set of related algorithms for distributed caching are proposed and studied in [Sinnwell and Weikum 1997]. These algorithms use estimates of cost and benefit to determine a good placement of (possibly) replicated items in a distributed system. Cost-based replacement policies have also been proposed in [Acharya et al. 1995] and [Cao and Irani 1997]. This work, however, cannot be applied well to distributed query processing as it is geared towards environments in which clients request *individual* objects. Thus, the execution model assumed in these other studies is much simpler than that of a distributed query processor. A single query can involve access to *many* objects (tables and indices) and there are many different ways to execute a query so that query optimization is required. Again, this is why cache investment is integrated into query optimization, something that is not considered in this previous cost-based work.

In order to estimate the benefit of caching, both WATCHMAN and the work described in [Sinnwell and Weikum 1997] propose to estimate the frequency of accessing an object as

$$\frac{k}{\text{now} - \text{last\_access}(k)}$$

where *now* is the current time and *last_access(k)* is the timestamp of the k'th latest reference to the object. The formula could also be used instead of "reference counting with aging" in order to implement a variant of the Reference-Counting policy. This variant would have higher overhead and result in better frequency estimates; this variant, however, would not help to improve the performance of the Reference-Counting policy for mixed query workloads or in the presence of heterogeneous servers; i.e., the situations in which the Reference-Counting policy performs poorly.

Data placement has of course also been studied in the context of parallel database systems [Copeland et al. 1988]. A dynamic approach to adjust the data placement in a parallel disk system was proposed in [Scheuermann et al. 1998]. Again, none of these approaches integrate data placement and query optimization nor do they consider the execution of sub-optimal plans in order to improve the data placement. Furthermore, the performance tradeoffs of parallel and distributed systems are different (consider, for example, communication costs).

Another related technique is *multi-query optimization* as proposed in [Sellis 1988]: like cache investment, multi-query optimization aims to improve the performance of a whole *set* of queries. The major difference between cache investment and multi-query optimization is that cache investment optimizes one query at a time whereas multi-query optimization optimizes a whole set of queries at once. This difference becomes clearer if we look at the situations in which cache investment and multi-query optimization are applicable. The purpose of cache investment is to improve a *stream* of queries submitted at a client: think, for example, of an online-user that iteratively asks queries; the user will ask one query at a time, every query of the user will be optimized individually, and cache investment takes effect to make the right caching decisions if the user happens to, say, ask particularly often for sales information in North Carolina. Multi-query optimization is applicable to optimizing the queries of several concurrent users at once; that is, multi-query optimization will optimize a set of queries issued by concurrent users and try to avoid duplicate work if several concurrent users ask for, say, the same sales information. Multi-query optimization and cache investment are, thus, complementary techniques which can both be used to improve the performance of a distributed database system.

8. CONCLUSIONS

We began this paper by identifying the relationship 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 showed how the query optimizer can be extended with *cache investment* so that it produces good query execution plans and implicitly makes long-term caching decisions at the same time. We showed that this extension is possible without changing basic components of the query optimizer. Furthermore, this paper presented and evaluated alternative policies which determine when and for which data the investment required to initiate caching should be paid.

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 (i.e., data shipping or query shipping or hybrid shipping without cache investment). 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. In our study, these policies were used as baselines for the evaluation of the other policies.

The Reference-Counting and Profitable policies take a *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 in many situations significantly better performance than the Conservative and Optimistic policies. 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: in the presence of "mixed" query and update workloads or heterogeneous servers, the system should employ the Profitable policy; other systems should employ the Reference-Counting policy because it is easier to implement and has less overhead.

There are many directions in which this work can be extended. 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, and we plan to study cache investment for architectures that support query result caching as well. Also, studying cache investment in a system that employs multi-query optimization in order to optimize the queries of concurrent clients in one batch is a promising avenue for future work. Moreover, we would like to study *hybrid* cache investment policies that combine the advantages of the Reference-Counting and Profitable policies; the idea is to apply the expensive Profitable policy only if this is necessary and otherwise rely on the cheap Reference-Counting policy. Furthermore, we intend to study how the policies presented in this work can be combined with techniques for global memory management such as those proposed in [Franklin et al. 1992]; here, the goal is to make good caching decisions for all (or a large group of) clients rather than for every client individually. Finally, we would like to study cache investment in the presence of proxy caching or more general caching hierarchies.

Acknowledgments

We would like to thank Björn Thór Jónsson who designed and implemented the simulator used in the performance experiments. Furthermore, we would like to thank Markos Zaharioudakis for providing us with his index-caching SHORE code. We would also like to thank Laurent Amsaleg and the anonymous referees for many useful comments.

REFERENCES

Acharya, S., Alonso, R., Franklin, M., and Zdonik, S. 1995. Broadcast disks: Data management for asymmetric communication environments. In *Proc. ACM SIGMOD Conf.* (San Jose, CA, USA, May 1995), pp. 199–210.

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

CAO, P. AND IRANI, S. 1997. Cost-aware WWW proxy caching algorithms. In *Proc. of USENIX Symp. on Internet Technology and Systems* (Dec. 1997), pp. 193–206.

CAREY, M., DEWITT, D., FRANKLIN, M., HALL, N., MCAULIFFE, M., NAUGHTON, J., SCHUH, D., SOLOMON, M., TAN, C., TSATALOS, O., WHITE, S., AND ZWILLING, M. 1994. Shoring up persistent applications. In *Proc. ACM SIGMOD Conf.* (Minneapolis, MI, USA, May 1994), pp. 383–394.

CAREY, M. AND LU, H. 1986. Load balancing in a locally distributed database system. In *Proc. ACM SIGMOD Conf.* (Washington, USA, 1986), pp. 108–119.

CHAUDHURI, S. AND NARASAYYA, V. 1997. An efficient, cost-driven index selection tool for microsoft SQL server. In *Proc. of the VLDB Conf.* (Athens, Greece, Aug. 1997), pp. 146–155.

CHEN, C. AND ROUSSOPOULOS, N. 1994. 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).

COPELAND, G., ALEXANDER, W., BOUGHTER, E., AND KELLER, T. 1988. Data placement in bubba. In *Proc. ACM SIGMOD Conf.* (Chicago, IL, USA, May 1988), pp. 99–108.

CUREWITZ, K., KRISHNAN, P., AND VITTER, J. 1993. Practical prefetching via data compression. In *Proc. ACM SIGMOD Conf.* (Washington, DC, USA, May 1993), pp. 43–53.

DAR, S., FRANKLIN, M., JÓNSSON, B., SRIVASTAVA, D., AND TAN, M. 1996. Semantic data caching and replacement. In *Proc. of the VLDB Conf.* (Mumbai, India, Sept. 1996), pp. 330–341.

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

EFFELSBERG, W. AND HÄRDER, T. 1984. Principles of database buffer management. *ACM Trans. on Database Systems 9*, 4, 560–595.

FRANKLIN, M. 1996. *Client Data Caching: A Foundation for High-Performance Object Database Systems.* Kluwer Academic Press.

FRANKLIN, M., CAREY, M., AND LIVNY, M. 1992. Global memory management in client-server DBMS architectures. In *Proc. of the VLDB Conf.* (Vancouver, Canada, 1992), pp. 596–609.

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

FRANKLIN, M., JÓNSSON, B., AND KOSSMANN, D. 1996. Performance tradeoffs for client-server query processing. In *Proc. ACM SIGMOD Conf.* (Montreal, Canada, June 1996), pp. 149–160.

FRANKLIN, M. AND KOSSMANN, D. 1997. Cache investment strategies. Technical Report CS-TR-3803 (May), University of Maryland, College Park, MD 20742.

GANGULY, S., HASAN, W., AND KRISHNAMURTHY, R. 1992. Query optimization for parallel execution. In *Proc. ACM SIGMOD Conf.* (San Diego, USA, June 1992), pp. 9–18.

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

HOROWITZ, E. AND SAHNI, S. 1976. *Fundamentals of Data Structures.* Computer Science Press, Rockville, MD, USA.

HUANG, Y., SISTLA, P., AND WOLFSON, O. 1994. Data replication for mobile computers. In *Proc. ACM SIGMOD Conf.* (Minneapolis, MI, USA, May 1994), pp. 13–24.

IOANNIDIS, Y. AND KANG, Y. 1990. Randomized algorithms for optimizing large join queries. In *Proc. ACM SIGMOD Conf.* (Atlantic City, USA, April 1990), pp. 312–321.

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

Keller, A. and Basu, J. 1994. A predicate-based caching scheme for client-server database architectures. In *Proc. of the IEEE Conf. on Parallel and Distributed Information Systems* (Sept. 1994), pp. 229–238.

Kemper, A., Kossmann, D., and Matthes, F. 1998. SAP R/3: a database application system. Tutorial handouts for the *ACM SIGMOD Conference on Management of Data*, Seattle, USA.

Kraiss, A. and Weikum, G. 1998. Integrated document caching and prefetching in storage hierarchies based on markov-chain predictions. *The VLDB Journal 7*, 3 (Aug.), 141–162.

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

Lohman, G. 1988. Grammar-like functional rules for representing query optimization alternatives. In *Proc. ACM SIGMOD Conf.* (Chicago, IL, USA, May 1988), pp. 18–27.

Mayr, T. and Seshadri, P. 1999. Client-site query extensions. In *Proc. ACM SIGMOD Conf.* (Philadelphia, PN, USA, June 1999), pp. 347–358.

O'Neil, E. J., O'Neil, P. E., and Weikum, G. 1993. The LRU-K page replacement algorithm for database disk buffering. In *Proc. ACM SIGMOD Conf.* (Washington, DC, USA, May 1993), pp. 297–306.

Palmer, M. and Zdonik, S. 1991. FIDO: A cache that learns to fetch. In *Proc. of the VLDB Conf.* (Barcelona, Sept. 1991), pp. 255–264.

Rodgriguez-Martinez, M. and Roussopoulos, N. 2000. MOCHA: a self-extensible database middleware system for distributed data sources. In *Proc. ACM SIGMOD Conf.* (Dallas, TX, USA, May 2000), pp. 213–224.

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

Scheuermann, P., Shim, J., and Vingralek, R. 1996. Watchman: A data warehouse intelligent cache manager. In *Proc. of the VLDB Conf.* (Mumbai, India, Sept. 1996).

Scheuermann, P., Weikum, G., and Zabback, P. 1998. Data partitioning and load balancing in parallel disk systems. *The VLDB Journal 7*, 1 (Feb.), 48–66.

Schkolnick, M., Finkelstein, S., and Tiberio, P. 1988. Physical database design for relational databases. *ACM Trans. on Database Systems 13*, 1 (March), 91–128.

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

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

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

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

Stonebraker, M., Jhingran, A., Goh, J., and Potamianos, S. 1990. On rules, procedures, caching and views in data base systems. In *Proc. ACM SIGMOD Conf.* (Atlantic City, USA, April 1990), pp. 281–290.

Wolfson, O. and Jajodia, S. 1992a. An algorithm for dynamic data distribution. In *IEEE Workshop on Management of Replicated Data* (Nov. 1992).

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

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

Yao, S. 1977. Approximating block accesses in database organizations. *Communications of the ACM 20*, 4 (April), 260–261.

Zaharioudakis, M. and Carey, M. 1997. Highly concurrent cache consistency for indices in client-server database systems. In *Proc. ACM SIGMOD Conf.* (Tucson, AZ, May 1997), pp. 50–61.