Submitted to the 1998 VLDB Conference.

Cache Investment for Indexes

Gerhard Drasch[1] Donald Kossmann [1*] Michael J. Franklin[2†]

[1]Universität Passau
FMI
D-94030 Passau

[2]University of Maryland
Department of Computer Science
College Park, MD 20742

Draft - February 20, 1998

Abstract

Modern distributed information systems depend on caching to dynamically place data close to where it is needed. Caching can have a dramatic effect on the performance of distributed queries. Likewise, query execution plans dictate the shipment of data and hence, help determine what can be cached where. Because of this interdependency, performing query optimization without concern for caching can have disastrous consequences for both short-term and long-term performance. *Cache investment* extends the query optimizer of a distributed database system to account for such concerns. In this paper, we examine how cache investment can be used to make the right decisions about the caching of indexes. We develop two alternative cache investment policies for indexes and implement them in the SHORE database system. Using this implementation, we conduct a detailed study of the performance tradeoffs of investment for indexes. The experimental results show that cache investment for indexes can improve the performance of applications by as much as a factor of ten compared to traditional distributed query processing techniques such as query shipping or data shipping.

1 Introduction

Most data access today is performed in a distributed, client-server environment. Applications are run at client machines, and these applications access data that are stored at local and/or remote server machines. Examples of such environments include traditional relational and object-relational databases, object-oriented databases (OODB), and world-wide-web browsers. In client-server systems, *data caching* has the potential to provide tremendous performance improvements by allowing data to be dynamically replicated and placed close to the applications that use them. Such dynamic placement can reduce communication costs, shorten path-lengths for data access, and store pre-computed results, all while adjusting to changes in access patterns.

The benefits of caching have long been recognized and exploited in the OODB and WWW environments. Relational and Object-Relational database systems, however, have largely ignored caching

*Contact Author; email: `kossmann@db.fmi.uni-passau.de`

†Michael Franklin's 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 Bellcore, and by an IBM SUR grant.

issues. This deficiency has driven some database *application* systems to implement their own caching in *middleware*, on-top of the database system. One example of such a system is SAP R/3, which is the market leader for business administration systems and one of the most important database application systems these days. SAP R/3 has a three-tier architecture in which the top tier carries out presentation services, the middle tier carries out the SAP R/3 application programs, and the back-end stores the database [BEG96, DHKK97]. In an SAP R/3 installation, the back-end is managed by a standard relational database product (e.g., Oracle, DB2, etc.), while base data are cached and (primary) indexes on those data are created in the SAP R/3 middle tier.

1.1 The Need for Cache Investment

Forcing application systems to implement their own data and index caching on-top of the database system introduces complexity and inefficiencies. For example, since SAP implements its own caching, query processing in SAP is not fully supported by the DBMS's optimizer. This places an undue burden on SAP application programmers, requiring them to act as "query optimizers". For example, the programmer of an SAP application program must decide whether a particular join should be executed in the middle-tier or by the database back-end server. The SAP programming interface provides different constructs for implementing joins and these constructs imply where operations are carried out. Furthermore, SAP programmers and SAP system administrators must determine which data are cached in the middle-tier: the administrator must mark relevant data as cacheable, and the programmer must write programs in such a way as to exploit the cache.

Decades of database systems development effort have been aimed at relieving database users of the burdens of query optimization. Query optimization technology, however, has not yet caught up to the rapid proliferation of distributed data access. As evidenced by the SAP system, the result is that users who wish to exploit the benefits of caching must currently manage such caching themselves, and often must fight the underlying database system in order to do so.

We have proposed *Cache Investment* as a technique for integrating caching knowledge and decision making into a query optimization framework [FK97]. Cache investment is based on the realization that there is a circular dependency between query optimization and caching. That is, the best execution plan for a distributed query will depend on the location of cached copies of relevant data, but the location of those copies is determined by the way that queries are executed (as sites can cache data only if those data are sent to them). Cache investment breaks this circularity by taking a broader view of query optimization in the presence of caching in order to consider what data are or should be cached at a client. In contrast to traditional approaches, cache investment will sometimes advise the optimizer to generate a suboptimal plan for a particular query in order to load the cache of a client with relevant data. The extra work for that query is an *investment* that will pay off in reduced response times for future queries initiated at that client.

1.2 Investing in Indexes

As originally proposed, the notion of cache investment did not take indexes into account. Indexes are key to reducing the amount of data that must be sent from servers to clients, and thus, caching them can greatly improve the performance of a client-server database system. Caching indexes at clients, however, imposes different demands on a cache investment strategy than regular data for several reasons. First, most queries cannot be answered using indexes alone. As such, indexes must compete for space in the cache with the data that they reference. Second, indexes are updated in a different manner and with different frequency than base data. Finally, indexes can be reconstructed from the base data, and therefore, can either be imported from a server or constructed on the fly at the client.

In this paper, we describe the cache investment framework and show how it can be extended to deal with indexes. We present two approaches to cache investment for indexes and study their performance by implementing them in the SHORE database system [CDF+94]. Using a workload based on the benchmark used by Zaharioudakis and Carey to study alternative concurrency control protocols for indexes in client-server systems [ZC97], we show that our approach outperforms the best "traditional" approaches by as much as a factor of ten in some situations.

We also extend our results to a particular kind of index that we call a *private index*. Private indexes are indexes that a client constructs locally to describe data that are of interest to that client. Private indexes are an alternative to obtaining and caching parts of existing indexes from the servers. Private indexes are attractive in cases where the server does not maintain an index that would be helpful to a particular client, or in systems where indexes cannot be cached because the servers do not provide support to maintain the consistency of cached indexes at clients (e.g., in legacy or heterogeneous database systems). We describe extensions to the cache investment framework to decide when to construct and destroy private indexes at clients.

The remainder of this paper is structured as follows: In Section 2, we give an overview of alternative ways that existing systems process queries and demonstrate how our techniques can be used to improve all these systems. In Section 3, we present the cache investment approaches in more detail, focusing on how the approaches deal with indexes. Section 4 describes the experimental environment and Section 5 presents the results of the performance experiments. Section 6 shows how cache investment can be applied to private indexes. Section 7 presents our conclusions.

2 Architectural Issues

2.1 Hybrid Shipping

Query processing in today's client-server environments is typically based on either a query-shipping or a data-shipping architecture. In a query-shipping system, all operators of a query (scan, sort, join, group-by, etc.) are executed at the server machines that store the data used in the query. In a data-shipping system, all operators of a query are executed at the client machine that initiated the query. The benefits of query shipping include: the reduction of communication costs for high selectivity queries, the ability to exploit server resources when they are plentiful, and the ability to tolerate resource-poor (i.e., low cost) client machines. In contrast, data-shipping has the advantages of exploiting the resources (CPU, memory, and disk) of powerful client machines and reducing communication in the presence of locality or large query results. Both of these factors allow data-shipping to scale as users, and hence, their desktop resources, are added to the system.

Neither data-shipping nor query-shipping is the best policy for query processing in all situations. Thus, a system that supports only one of these policies is likely to have sub-optimal performance

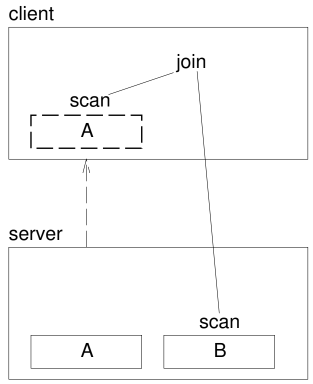

Figure 1: Hybrid Shipping (HY)

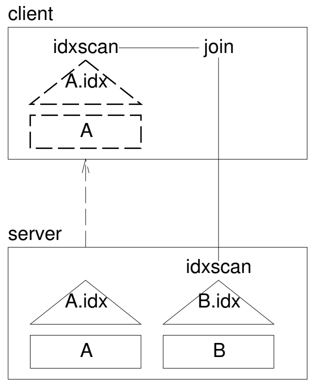

Figure 2: HY with Indexes

under certain workload and/or system conditions. In previous work, we have shown that for complex queries, a *hybrid-shipping* (HY) approach can often outperform both pure policies [FJK96]. Hybrid-shipping provides the flexibility to execute query operators on client and server machines, and allows the caching of data at clients. The approach is illustrated in Figure 1; in the figure, the join and the `scan(A)` operators are executed at the client whereas the `scan(B)` operator is carried out at the server. Note that the `scan(A)` operator will use the client's cache as much as possible, bringing to the client only those parts of A that are not in the cache already. In contrast, the `scan(B)` operator neither uses nor changes the state of the client's cache.

Obviously, hybrid-shipping has the potential to perform as well or better than either data shipping or query shipping; in fact, data and query shipping can be seen as special cases of hybrid shipping. To take the best advantage of hybrid shipping, however, requires extending the query optimizer of a system. The optimizer of a hybrid-shipping system must decide where (at the client or the server) every query operator is carried out in addition to all the decisions the optimizers of a query-shipping or a data-shipping system must make.

The decisions made by the optimizer also dictate what data can be cached at a client. In the example of Figure 1, a copy of Table A may be retained at the client after the query has been executed as a by-product of placing the `scan(A)` operator at the client. This plan might be beneficial if the client frequently initiates queries involving Table A, but it will most likely be a poor plan if the client frequently accesses Table B but not Table A. Thus, because of its impact on caching, query optimization in a hybrid-shipping environment can impact the response time of *future* queries, in addition to that of the particular query being optimized. Traditional optimizers have not been designed to factor in the effects of a query evaluation plan on the performance of queries that might arrive in the future; in particular, traditional query optimizers have not been designed to make special decisions in the presence of caching. Cache investment is an approach that extends traditional optimizers to become cache-aware in a hybrid-shipping system.

2.2 The Role of Indexes

Indexes add further options for query processing in a hybrid-shipping system. In a query-shipping system, indexes can be established and used to speed-up the execution of queries at the server machines in a straightforward way. In a data-shipping system, indexes would typically also be established at servers, but they could be used and possibly cached at client machines. Comparing query and data shipping, the presence of indexes makes data shipping somewhat more attractive because they allow clients to request only the relevant items (i.e., data or index pages) from the server. Hybrid-shipping systems have the flexibility to access indexes at servers or at clients. Of course, the optimizer of such a system must take into account the caching of indexes in addition to base data.

Figure 2 shows an example hybrid-shipping plan for a simple join query that involves two indexes and two tables. The optimizer places the `idxscan(A.idx)` operator at the client so that the relevant pages of Table A and Index A.idx will be brought to the client's cache if they are not there already. This plan is good if A and A.idx are frequently used in queries at the client. An alternative plan would place both idxscan operators at the client, which would be beneficial if the client's cache were large enough to hold all relevant pages of both tables and indexes and all such pages were frequently used at the client. Another alternative would be to carry out an ordinary `scan(A)` operator (without the index) at the client. This plan would be beneficial if Table A were involved in many queries at the client, but A.idx were not useful for most of these queries. As these examples demonstrate, the presence of indexes opens up a large variety of options for hybrid shipping and caching. In this paper, we show how cache investment can be used to extend a query optimizer to take all these options into account and make the right decisions.

2.3 Cache Investment Basics

The main task of cache investment is to determine which data items (tables, horizontal partitions of tables, and indexes) should be cached at a client. Cache investment accomplishes this task by keeping usage statistics for every data item at each client and assigning a value $V_d(c)$ to every data item d at client c based on these statistics. The value of a data item at a client indicates how useful it is to cache that item at the client: if, for example, a data item is frequently used at a client, then it is likely to have a high value at that client. Given these values, cache investment makes the following two decisions:

1. Which data items are worth caching? There is a cost or investment, $I_d(c)$, associated with bringing a data item d (or parts of the item) into client c's cache. Only those data items whose value is higher than the investment to bring the data item into the cache (i.e., $V_d(c) > I_d(c)$) are worth caching.

2. Since the size of a cache is limited, which of the worthy data items should be cached? Essentially, this is an (NP-hard) knapsack problem in which the goal is to maximize the sum of the values of all cached data items at every client; to find a good solution, the obvious heuristics of packing those data items with the highest $V_d(c)/size$ ratio can be used [HS76].

So, just as an investor on Wall Street decides which kinds of stock to keep in his/her portfolio, cache investment decides which data items to buy (i.e., bring into a client's cache), and the value of a data item corresponds to the expected return on investment of a share.

Figures 3 and 4 illustrate how cache investment can be integrated into an existing client-server database system. In a first step (Figure 3), we make the optimizer cache-aware. Such a cache-aware optimizer asks the client's cache manager about the current state of the cache and based on this

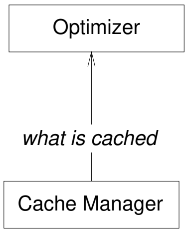

Figure 3: Cache-aware Optimizer

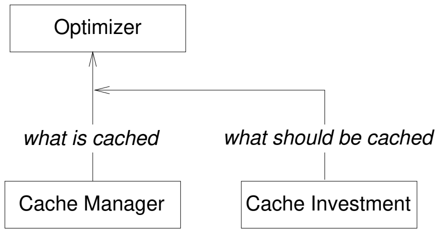

Figure 4: Integrating Cache Investment

information generates an execution plan (including operator placement) for a query. If, for example, large portions of Table A are cached in the client's (main memory) cache, then the optimizer would most likely place a `scan(A)` operator at the client; otherwise, the optimizer would place the `scan(A)` operator at the server.

Making an optimizer cache-aware is quite simple and, in fact, distributed query optimizers such as the System R* optimizer provide all the required functionality [Loh88]. Unfortunately, making an optimizer cache-aware is not enough. Initially, all client caches are empty. In this case, a cache-aware optimizer would likely place all scan operators at servers. Thus, client caches would remain empty, and caching would never be exploited. Cache investment must be integrated with the optimizer to avoid this problem. Using cache investment (as shown in Figure 4), when the cache-aware optimizer requests the current status of the cache, the answer it receives is augmented to include what the cache investment policy determines *should be* cached. In effect, cache investment lies to the optimizer, so that it believes that all the data items that should be cached are already present in the cache. The optimizer is therefore likely to place scan operators at the client, which will result in the desired items being brought to the client and possibly cached there. In this way, cache investment can be integrated into the cache-aware optimizer without having to change its search strategy, cost model, etc.

Obviously, the optimizer of Figure 4 will not always generate optimal plans for a particular query instance. It can generate a suboptimal plan whenever it places a scan operator at a client in the false belief that the corresponding data are already present in the client cache. This behavior, however, is intentional; the difference in cost between this suboptimal plan and an optimal plan is the cost of the *investment* for bringing the data item into the client's cache. Initiating caching as a by-product of executing a suboptimal query plan is likely to have lower overhead than establishing a new replica of data in a distributed database system like Mariposa [SAL+96]. This is because with the cache investment approach, a great deal of the IO required to read the data item from the server's disk and communication costs to ship results and pages of the data item to the client are amortized by the fact that at least some of the work would have had to be done in order to execute the query in any case. In contrast, systems that treat replication separate from query execution incur the entire cost of replication as pure overhead.

Using this basic mechanism of cache investment, a number of different investment *policies* can be defined. These policies differ in the way that they assign values to data items and estimate the investment to get these data items. Next, we describe policies for performing cache investment on base data and indexes.

3 Cache Investment Policies

A cache investment policy dictates how the value of a data item at a client ($V_d(c)$) is determined based on statistics of data item usage at that client, and it also dictates how the investment to bring a data item to the client ($I_d(c)$) is determined. In this section, we first describe cache investment policies for base data in the absence of indexes. We then describe how these policies can be extended to cope with indexes. What makes cache investment for indexes interesting is that the presence of indexes opens up a wide variety of alternatives for caching indexes, tables, or both at clients and also impacts the model used to determine which items are worth caching. As will be seen, however, the basic mechanisms of cache investment can be naturally extended to account for indexes.

3.1 Cache Investment for Tables

We have developed two policies for determining the value of and the investment to get a table or a horizontal partition of a table at a client in the absence of indexes. These are called the profitable and the reference counting policies. The way the two policies determine the value and investment of a table is very different, but both policies have in common that they compute a benefit $v_{i,d}(c)$ of caching a table for a specific query i at client c and compute the $V_d(c)$ as the weighted sum of all $v_{i,d}(c)$'s. More specifically, $V_d(c)$ is computed from the $v_{i,d}(c)$'s using a technique called *periodic aging by division* that was originally proposed for database buffer replacement [EH84].

If n queries have been processed at client c, then $V_d(c) = \sum_{i=1}^{n} \alpha^{n-1} * v_{i,d}(c)$ or putting it differently, $V_d(c) := \alpha * V_d(c) + v_{n,d}(c)$ after executing query n at the client. α is a tuning parameter of this approach. We have found that a setting of $\alpha = 0.9$ usually provides good performance for cache investment, so we use that value throughout this paper. A sensitivity analysis of the policies to the setting of α can be found in [FK97].

In the following, we describe the profitable and reference counting policies in the no-index case and show how cache investment deals with updates in the no-index case. Due to space considerations, however, we keep the exposition short; the details of the no-index case can be found in [FK97].

3.1.1 The Profitable Policy for Tables

The profitable policy is based on an approach that has been used for the physical design of relational database systems [SFT88, CN97]). The idea is to do a "what-if" analysis and compute $v_{i,d}(c)$ as the reduction in response time of query i if table d is cached at the client prior to the execution of query i. That is, for every table used in the query, the profitable policy generates two plans using the system's query optimizer. The first plan is generated telling the optimizer that the whole database, including this table, is cached, and the second plan is generated telling the optimizer that the whole database, except this table, is cached. The optimizer's cost model will assign a cost to both plans (of course, the cost of the first plan should be less than or equal to the cost of the second plan); $v_{i,d}(c)$ is set to be the difference between the costs of the two plans.

As an example of this type of "what-if" analysis, consider a query that asks for all `Employees` with `salary` $>$ `100,000`. The cost of the first plan is, say, 10 seconds for scanning through the whole `Employee` table in the client's main memory. The cost of the second plan could be, say, 200 seconds for scanning through the whole `Employee` table at the server (possibly involving disk IO) and shipping the query result from the server to the client. In this example, the value of the `Employee` table at the client would be increased by 190 secs.

With a similar kind of "what-if" analysis, the profitable policy determines the investment required to bring table d into client c's cache as a by-product of executing query i in order to do the $V_d(c) > I_d(c)$ test. The profitable policy computes this investment as the difference in cost of the suboptimal plan for query i that brings the table into the cache and an optimal plan for query i. Note that the investment, therefore, depends on characteristics of query i such as the selectivity of predicates. Explicitly computing this investment for every query gives the profitable policy the opportunity to wait for those queries for which the investment is acceptably small.

3.1.2 The Reference Counting Policy for Tables

The profitable policy does a very good job in determining which tables should be cached at a client and deciding when these tables should be brought to the client. The profitable policy, however, has a very high overhead because it requires the generation of several plans for each query. In fact, the profitable policy generates $2 * t + 2$ plans for a query that involves t tables. We note that these plans can be generated very quickly (e.g., using greedy algorithms or simple transformations on an existing plan) because these plans are not executed but, nevertheless, the overhead of the profitable policy can be quite significant for queries that involve many tables. The goal of the reference counting policy is to make nearly as good decisions as the profitable policy with far less overhead.

The reference counting policy works by simply setting $v_{i,d}(c) = 1$ if table d is used in query i at client c and $v_{i,d}(c) = 0$, otherwise. In order to do the $V_d(c) > I_d(c)$ test and decide whether table d is worth caching, the reference counting policy takes a tuning parameter (T), sets $I_d(c) = T$ for all tables and at all times, and decides that d is worth caching if $V_d(c) > T$. If T is poorly set, then the reference counting policy can degenerate to behave like pure data shipping or like pure query shipping. For the workloads we study in this paper, our experiments have shown that $T = 2.0$ is a good setting.[1]

There are many types of workloads in which the reference counting policy is likely to assign values to tables in such a way that makes cache investment less effective than with the profitable policy (we describe one such workload in Section 5). However, as with many query optimization cost prediction issues, the goal is not necessarily to be completely accurate. Rather, it is important to make fast decisions that accurately distinguish the good options from the truly bad ones. Thus, the reference counting policy is a viable, low-overhead candidate for a cache investment policy.

3.1.3 Dealing with Updates

An important feature of cache investment is that it 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 nicely be 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. Since we implemented cache investment in the SHORE database

[1]The sensitivity of T is analyzed in [FK97]. For space reasons, we do not discuss the tuning of this parameter any further in this paper.

system and since SHORE is based on callback locking, we briefly describe how cache investment considers updates in this particular environment [CDF+94, Fra96].[2] Before a page (or object) is modified in SHORE, the SHORE server that owns that page sends a callback message to all clients that have a copy of that page in their cache. As a result, all those clients will give up their copy of the page as soon as all running transactions that read that page have committed, and they will inform the server that they have done so. 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 d are cached at client c, and k pages of that table are called back, then the value of the table is adjusted as follows:

$$V_d(c) := \frac{a-k}{a} * V_d(c)$$

3.2 Cache Investment for Indexes

In the previous section, we showed how cache investment decides which base data should be cached at a client. In this section, we extend our model to account for the presence of indexes. Basically, an index can be seen as a type of *data item* so "what-if" analyses and reference counting are applicable for making caching decisions. There are, however, three important points that must be kept in mind:

- An index scan reads only a small fraction of an index. Thus, we need to find ways to predict how much cache space would actually be required to cache the used portions of an index because the *size* or *required space* of a data item is an important factor for cache investment (i.e., to see how many of the worthy data items fit in the cache). Note that we need to *predict* the required space to cache an index and cannot wait and see how much space the index uses once it is cached because cache investment must decide which data items to cache *before* they are actually cached. Also note that predicting the used portions of a base table was not an issue in the previous section because a table scan always reads the *whole* table in the absence of indexes.

- It is usually not advantageous to only cache index pages; the referenced pages of the base tables should also be cached. As a result, we need to consider index pages plus base table pages when we estimate the space requirements of cached indexes.

- At the same time, however, indexes and base tables should be considered as separate data items because indexes and base tables compete for space in a client's cache. In the long run, cache investment must decide whether indexes (plus the used base table pages only), whole base tables, or both should be cached.

In the following, we first describe how we estimate the space occupied by an index in the cache. We then give several examples and describe how the presence of indexes affects the reference counting and profitable policies. Finally, we show how updates are dealt with in the presence of indexes.

3.2.1 Space Requirements of Cached Indexes

To estimate the space requirements of a cached 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

[2] In Section 6, we describe an alternative model to deal with updates for the Private Indexes protocol.

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 where 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 [Yao77].

Now, let us consider a situation in which we have estimated that all past queries involving an index could have been executed with S_i index pages and S_t base table pages. If the next query can be executed with i_{new} and t_{new} index and base table pages, then all past plus this next query can be executed using

$$S_i + i_{\text{new}} - \frac{S_i * i_{\text{new}}}{m}$$

index pages and

$$S_t + t_{\text{new}} - \frac{S_t * t_{\text{new}}}{n}$$

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 predictions that typically overestimate the actual space requirements.

It should be noted that some queries can be executed using an index and without accessing the base table at all; an example is a query that asks for the average salary of the top 10,000 `Employees` and which can be executed using an `Emp.salary` index only. In this case $t_{\text{new}} = 0$ and the formulae above remain applicable.

3.2.2 Profitable Policy

Recall that the profitable policy is based on "what-if" analyses in order to find out which tables are worthiest to cache. In the presence of indexes, these "what-if" analyses are applied to every table involved in a query and 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 indexes, 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.

For a table, two plans are generated as part of the "what-if" analysis: Plan 1 is generated by telling the optimizer that the whole database including the table, but excluding all the indexes on that table, are cached; this plan is likely to have a table scan operator at the client, assuming that the table can cheaply be read from the client's main memory, but no indexes are available at the client. Thus, Plan 1 is exactly the plan that would use the cached copy of the whole table, if the whole table is cached. Plan 2 is generated telling the optimizer that the whole database, excluding that table and all the indexes on that table, is cached; this plan is likely to be a query shipping plan that carries out all operators on that table in the best possible way at the server (with or without indexes). The benefit of caching the whole table (i.e., $v_{i,d}$) is then computed as the difference in cost between the second and the first plan.

For an index, the following two plans are generated: Plan 1 is generated telling the optimizer that the whole database including that index and the corresponding table, but excluding all other indexes on that table, are cached. This plan is likely to have an index scan operator at the client; that is, this is the plan that would use the cached copy of the index, if the relevant parts of the index and the base table are cached. Plan 2 is generated telling the optimizer that the whole database excluding that index, the corresponding table, and all other indexes on that table are cached; this plan is, again, likely to be a query shipping plan. The benefit of caching the index is then computed as the difference in cost between the second and the first plan. Similarly, "what-if" analysis are applied in order to compute the *investment* required to bring an index or a whole table to the client's cache and see whether an index or a table are worth caching.

To see how indexes and tables compete for space in the cache if the profitable policy is used, consider the following example involving an `Employee` table with $n = 100,000$ pages and an unclustered `Employee.salary` B$^+$-tree with $m = 1,000$ leaf pages. A client asks for all employees with salary greater than \$1,000,000, and let us assume that only very few employees qualify. In this example, the benefit of caching the whole table is likely to be 0 because a whole table scan at the client will be as expensive as or even more expensive than an index scan at the server plus the cost to ship the few result tuples to the client; so, the optimizer is going to generate a query shipping plan even if told that the whole table is cached at the client. On the other hand, the benefit of caching the index might be, say, a couple of seconds for savings in disk IO at the server and communication costs. Also, the *required space* (or size) of the index is going to be much smaller than that of the whole table because only a couple of pages of the index and table would need to be cached for the index plan whereas all 100,000 pages of the `Employee` table need to be cached in the table scan plan. The index, thus, has a much higher *value/size* ratio than the whole table. Furthermore, the investment to cache the index is going to be significantly smaller than the investment to cache the whole table. As a result, the profitable policy would advise the optimizer to cache the index rather than the whole table if many of these high selectivity queries involving the salary column are asked, and this is the right decision.

As another example consider a query that asks for all employees with salary greater than \$100, and let us assume that almost all employees satisfy that query. In this example, the benefit of caching the whole table is going to be huge. The benefit of caching the `Employee.salary` index together with all required pages of the `Employee` table is also large, but it is not going to be larger than caching the entire `Employee` table. Turning to the space requirements, these are going to be larger for caching the index than for caching the whole table in this example: caching the whole table requires $n = 100,000$ pages, while caching the index requires about 101,000 pages because the whole index (1000 pages) must be read in addition to the whole table. (Using the formulae of Section 3.2.1, $i_{new} \approx m$, so $S_i \approx m$; and $t_{new} \approx n$, so $S_t \approx n$.) In this example, thus, the table has a higher *value/size* ratio than the index. Furthermore, the investment to cache the whole table is significantly smaller than the investment to cache the index, as bringing the index to the client involves executing a large index scan with an unclustered index. As a result, the profitable policy would advise the optimizer to cache the table rather than the index in this example, and this is the right decision.

For query workloads that consist of mixes of high and low selectivity queries, the profitable policy accumulates the *values* of all indexes and tables, computes the space requirements to cache the indexes as described above, and decides which indexes and tables to cache considering the *value/size* ratio of indexes and tables individually. If the cache is large enough, it is, of course, also conceivable that cache investment decides that both an index and the whole corresponding table should be cached. To cache the `Employee.salary` index and the whole `Employee` table, the cache would, of course, have to be able to hold at least 101,000 pages.

3.2.3 Reference Counting Policy

The reference counting policy, as before, uses a simplified approach to determine which indexes 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 indexes 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 indexes and tables involved in the query plan. If, for example, the plan generated by the optimizer involves an index scan of the `Employee.salary` index, then the value of the `Employee.salary` index is incremented whereas the values of the `Employee` table and all other `Employee` indexes are not incremented. If, on the other hand, the plan involves an ordinary table scan, then the value of the `Employee` table is incremented whereas none of the values of the `Employee` indexes are incremented. As a result, the reference counting policy is able to make the right decisions in many situations. At the same time, the reference counting policy has almost no additional overhead because it calls the optimizer only once for every query. On the negative side, of course, the decisions of the reference counting policy cannot always be perfect in certain mixed workloads because the reference counting policy finds out which indexes 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. We discuss this effect in more detail in Section 5.2.

3.2.4 Dealing with Updates

As described in Section 3.1.3, the values of indexes and tables must be adjusted in the presence of updates. SHORE uses callback locking (i.e., invalidation) for cached pages of base tables and for cached leaf pages of indexes [ZC97] so we can continue to reduce the value of an index proportionally in the presence of updates. We must keep in mind, however, that we consider index d to be worth $V_d(c)$ only if all S_i relevant index pages and all S_t 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 k index and base table pages, the value of index d is, therefore, adjusted as follows:

$$V_d(c) := \frac{S_i + S_t - k}{S_i + S_t} * V_d(c)$$

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 indexes, but it is clearly not true for clustered indexes: for clustered indexes, an update that invalidates an index page always also invalidates one of the referenced cached base table pages. For clustered indexes, we therefore only take invalidations of base table pages into account and use the following formula to adjust the value of an index after k_t base table pages have been invalidated:

$$V_d(c) := \frac{S_t - k_t}{S_t} * V_d(c)$$

Another conservative assumption that we make in both formulae above is that we assume that everytime a base table page is invalidated, this page is actually one of the pages referenced by one of the cached pages of the index. This assumption is not true if we, for example, use several different indexes at the client so that a base table page is not referenced by the cache pages of all indexes. We could reduce the imprecision of this assumption by keeping track of which base table pages are referenced by which cached indexes, but we have not implemented this yet.

4 Experimental Environment

We implemented cache investment using the profitable and reference counting policies as described in the previous section, and we integrated both policies into the SHORE system in order to study their effectiveness in a traditional client-server database system. In this section, we briefly describe the benchmark we used and our implementation of cache investment. The results of the performance experiments are described in the next section.

4.1 Benchmark Specification

The performance of cache investment depends on a number of parameters: the processing power (CPU and disks) of the client and server machines, the size of the clients' caches and the servers' buffer pools, the network bandwidth, the frequency of updates, and the kind of queries (selectivity of predicates, joins, group-bys, order-bys, etc.) to name just a few. We varied all these parameters in our work in order to make sure that we observed all possible effects. For brevity, however, we show only a small number of experimental results and simply summarize the observations we made in the other experiments. The workloads used in the experiments shown here were derived from the workloads used by Zaharioudakis and Carey in their study on indexes in client-server database systems [ZC97]. 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 5. 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 so that every client has its own *private* partition that is only used in queries of that client and there is one *shared* partition which is used in queries of all clients. As shown in Figure 5, the HOTCOLD database has one unclustered B^+-tree index for every partition.

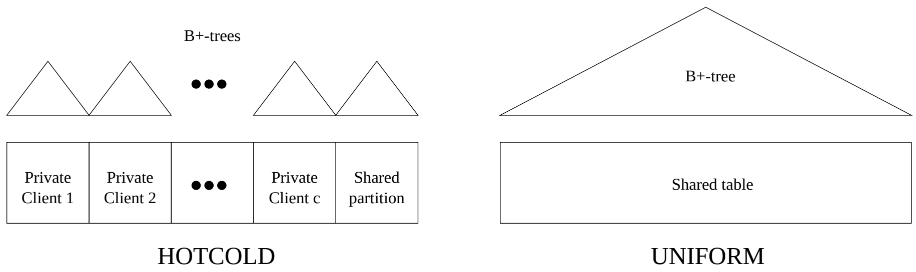

Figure 5: HOTCOLD and UNIFORM Databases

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

Table 1: Database Parameter Settings

Table 1 shows our parameter settings for the two databases. Both databases are about 40 MB (without indexes): 200,000 objects and 20 objects per 4 KB page. 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 called HOTCOLD and UNIFORM. In these workloads, every client issues a single stream of range queries and update operations, and in both workloads, we varied the frequency of update operations. A range query reads 100 objects using an unclustered index. In the HOTCOLD workload, 80% of the range queries of a client involve the client's private partition whereas 20% of the range queries involve the shared partition. In the UNIFORM workload, naturally, all queries involve the large (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 random object of the whole database (disregarding private and shared regions) and this update involves one insert and one delete operation on one of the indexes. In the original Zaharioudakis and Carey benchmark, update operations navigate through the database and modify objects on their way so that our update operations are more light-weight than theirs. However, both kinds of update operations modify objects and indexes.

We also studied a third workload that we call the POINTRANGE workload, which was not described in [ZC97]. This workload uses the HOTCOLD database and involves point and range queries; here, point queries read one object using an index and range queries read 100 objects using an index, as before. As an example for 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% in our experiments with the POINTRANGE workload. The settings of the most important parameters of all three workloads are summarized in Table 2.

	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 2: Workload Parameter Settings

4.2 Software and Hardware Used

We used Version 1.0 of SHORE, enhanced by code for client-side index caching which was described in [ZC97]; this code is not part of the official SHORE release.[3] A detailed description of SHORE can be found in [CDF$^+$94] or on the SHORE project's web pages (http://www.cs.wisc.edu/shore). SHORE provides state-of-the art implementations for record management, B$^+$-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 a query optimizer and basic relational query operators for joins, sorting, etc. In a client-server architecture, there is a question of where to optimize a query [HF86] 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 and the overheads of optimization were very small in our experiments.

As stated above, we ran experiments in many different configurations, including many different hardware configurations. In all experiments reported in this paper, the database was stored by a single, dedicated server machine and queries and updates originated from four concurrent client machines. The server was a SUN workstation 20/502MP with two 50MHz SPARC processors, 96MB of main memory, and a 4 GB disk drive. The four client machines were SUN workstations 10/20 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, and 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. To calibrate the cost model of the query optimizer, we ran a separate set of experiments in this environment.

5 Performance Experiments and Results

In this section, we present the results of some basic performance experiments with cache investment. The following four approaches are compared:

[3] Specifically, we used the *hybrid caching* approach of [ZC97].

1. hybrid shipping and cache investment with the profitable policy;
2. hybrid shipping and cache investment with the reference counting policy;
3. query shipping; as explained in Section 2.3, query shipping is, in these experiments, equivalent to a hybrid-shipping approach without cache investment;
4. data shipping, which carries out all query operators at clients and makes extensive use of caching pages of tables and indexes. Our data-shipping results are comparable to the best approach studied in [ZC97] (i.e., hybrid caching).

We first show results obtained for the HOTCOLD, POINTRANGE, and UNIFORM workloads. We then summarize the observations we have made in our other experiments.

5.1 HOTCOLD Workload

Figure 6 shows the average response time of the queries in the HOTCOLD experiments, varying the frequency of update operations.[4] The Figure shows clearly why cache investment in a hybrid-shipping architecture is needed to achieve acceptable performance: both data and query shipping show poor performance. This poor performance is due to either using the client resources too much (in the case of data shipping) or not at all (in the case of query shipping). Using query shipping, for instance, all queries have very high response time because they are executed at the server, the bottleneck of the system. Using data shipping, 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).

In contrast, hybrid shipping with cache investment shows significantly better performance using either the profitable or the reference counting policy here. Hybrid shipping outperforms query shipping because it executes all queries involving private data at the clients, thereby effectively using the clients' caches. Hybrid shipping outperforms data shipping because it executes all queries involving shared data at the server, thereby avoiding unnecessary overhead to move the shared data to the clients. In this experiment, the differences between the profitable and the reference counting policy are marginal for two reasons: First, 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 a single table and a single index.

Figure 7 shows the cost of the update operations in the HOTCOLD workload. Again, the two cache investment approaches have the best performance, data shipping is in the middle, and query shipping is the worst. Although query shipping 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 query shipping in this workload, even for update operations.

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

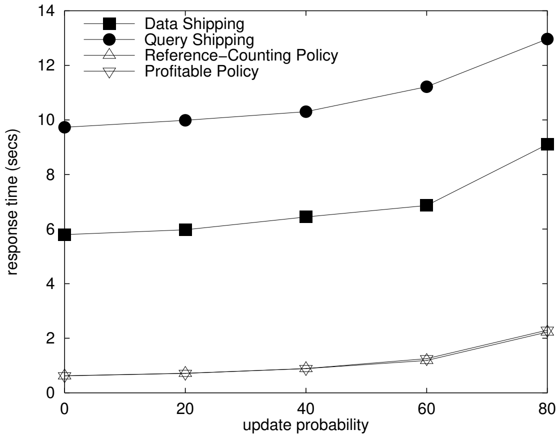

Figure 6: Resp. Time of Queries (secs)
HOTCOLD Workload

Cost of Updates	
Data Shipping	0.55 secs
Query Shipping	0.64 secs
Ref.-Counting Policy	0.38 secs
Profitable Policy	0.38 secs

Figure 7: Resp. Time of Update Ops (secs)
HOTCOLD Workload, 20% Update Prob.

5.2 POINTRANGE Workload

Figure 8 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 because the benefit of caching data that is mostly used in range queries is much higher than the benefit of caching data that is only used in point queries: point queries are cheap no matter where they are executed whereas range queries are only cheap if all the relevant index and base table pages can be found in the client's cache. Otherwise, range queries incur a significant amount of random disk IO at the server.

As can be seen in Figure 8, 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. Data shipping executes all queries at the clients and shows quite good performance for this workload because it achieves reasonably high hit rates in the client cache for the range queries on the private data. Data shipping still performs worse than the profitable policy, however. For example, when there are no updates, data shipping is outperformed by the profitable policy by a factor of two on average. As in the previous experiment, query shipping 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.

5.3 UNIFORM Workload

Figure 9 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. It would easily be possible to think of experiments with the UNIFORM workload in which data shipping is more attractive than query shipping by increasing the size of

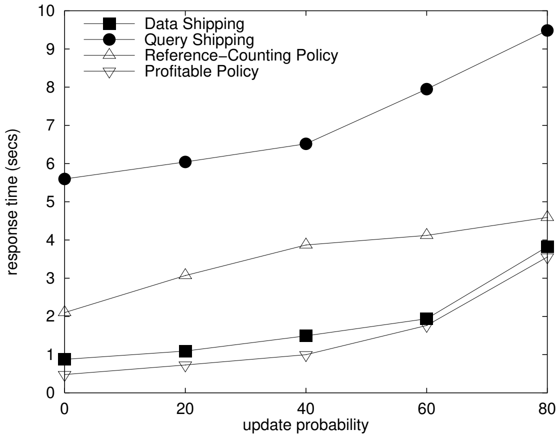

Figure 8: Resp. Time of Queries (secs)
POINTRANGE Workload

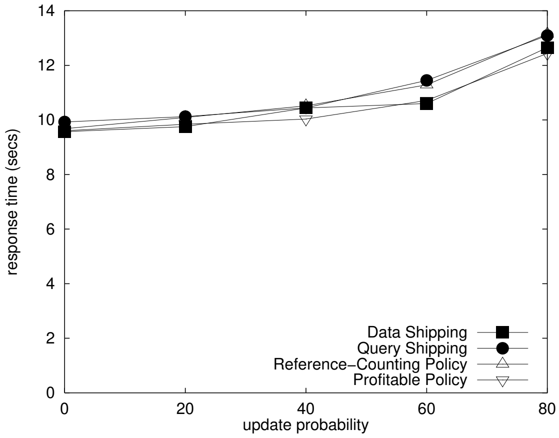

Figure 9: Resp. Time of Queries (secs)
UNIFORM Workload

the clients' caches or the number of clients. Likewise, it would be possible to think of experiments for which query shipping is more attractive than data shipping by reducing the number of clients or increasing the selectivity of the range predicate of the queries and reducing the size of the query results. The important point to notice is that with such a UNIFORM workload, hybrid shipping and cache investment perform no better than the other approaches. This is because cache investment basically mimics one or the other of the pure approaches in this case. Using the profitable policy, cache investment will naturally behave like the better of data and query shipping. Using reference counting, the behavior of cache investment in this case will depend on the setting of its tuning parameter, T. If T is set low (as in our experiments), the reference counting policy behaves like data shipping; otherwise, it would behave like query shipping.

5.4 Discussion and Summary

Figures 6, 7, 8, and 9 showed the results of three simple experiments that demonstrate the basic tradeoffs of cache investment. We carried out a large number of other experiments in which we varied the number of clients, the sizes of the clients' caches and of the server's buffer pool, the selectivity of predicates and workload mixes, but we essentially always observed the same effects: hybrid shipping with cache investment is the overall clear winner. There were situations (e.g., UNIFORM workloads) in which either data shipping or query shipping did as well as hybrid shipping with cache investment, but every time there was some skew in the data access or we had mixed workloads, pure data and query shipping were not flexible enough to treat all queries and all data in the best possible way. Also, data shipping and query shipping were never better than cache investment with the profitable policy.

Comparing the profitable and the reference-counting policies, the profitable policy was always at least as good as the reference-counting policy for the Zaharioudakis-Carey benchmark with its simple range and point queries, but the reference-counting policy showed very good performance in most situations as well: it was only workloads like the POINTRANGE workload where the statistics of the reference-counting policy were not detailed enough to make the right decisions. For workloads with more complex queries, our previous work [FK97] shows that the overhead of the profitable policy becomes significant (more than a second) for queries involving five or more tables: given the benefits that can be achieved, this overhead might, however, still be tolerable in many situations. In all, our experience indicates that both policies are viable candidates to implement cache investment with indexes.

6 Cache Investment for Private Indexes

In this section we shift focus slightly, to show how cache investment can be applied to make the right decisions for *private indexes*. A private index is an index that is established at a client and only used to execute queries initiated at that client. Private indexes are useful in situations in which the servers do not have the right indexes for certain clients or in database systems that do not support SHORE-style index caching (e.g., heterogeneous systems or databases in the Internet). Private indexes can be established at a client as a by-product of query processing. For example, consider a query that asks for all employees that work in a research department; i.e., `Employee` $\bowtie$ σ`(Dept)`. One way to process this query at a client would be to use an index join with, say, a main-memory hashtable on `Dept` (i.e., a classic hash join [Sha86]). If we build a hashtable for the whole `Dept` table rather than for the research `Depts` only, then we can keep this hashtable at the client for subsequent queries. So, cache investment can naturally be applied because we have a *benefit* for having the index at the client for future queries, and the *investment* would be to build a hashtable for the whole `Dept` table rather than for the research `Depts` only.

Both the profitable and the reference counting policies can be applied to decide which private indexes to construct at clients. The profitable policy would carry out "what-if" analyses as described in Section 3.2.2. The only complication is that the overhead of the profitable policy can become quite large in this case: rather than making "what-if" analyses for all *existing* indexes, the profitable policy must also make "what-if" analyses for all *conceivable* private indexes. For simple queries, we expect this overhead to be tolerable but for complex queries, the overhead can easily become prohibitive; finding good heuristics to narrow the search for useful private indexes is an important part of our future work. Turning to the reference counting policy, this policy always has low overhead, but it is slightly more complicated and error-prone for private indexes than for cached indexes and tables because after generating and executing a plan for a query, the reference counting policy must analyze the plan in order to determine which of all the possible private indexes (that do not yet exist) would have been helpful to improve the plan.

We implemented private indexes, hybrid-shipping and cache investment for private indexes in SHORE, and ran a number of experiments. We do not discuss in any detail here because the results are fairly straightforward: in most situations, the profitable and reference counting policies were able to make the right decisions and the wins compared to data shipping and query shipping were huge. One particularity of our implementation of private indexes is that we maintained private indexes by propagating updates rather than carrying out invalidations. As a result, we had to change the way cache investment reduces the value of a private index in the presence of updates: with every message that was sent to a client in order to propagate updates, the values of all indexes that were affected by the updates was reduced by the (constant) cost of sending an "update" message to the client, and cache investment explicitly destroyed a private index if the value of the private index dropped below

0. For comparison, recall from Section 3.1.3 that cache investment reduces the values of data items proportionally in an invalidation based scheme so that the value of a data item can never drop below zero. Changing the update model of cache investment is very easy, and this observation shows how well cache investment can be adapted to work in any kind of system.

7 Conclusion

In this paper, we showed how distributed query optimization can be extended with cache investment in order to generate good plans and make the right decisions about caching. The main focus of this paper was to show how cache investment can be applied to decide which indexes to cache and/or which indexes to construct at clients, if the right indexes do not yet exist or cannot be cached due to other constraints of the system. To this end, we devised two policies, called the profitable and the reference counting policies, and we implemented both policies in the SHORE distributed database system. We carried out a large set of experiments and found that both policies are in general good choices for cache investment for indexes. (Both policies have their own specific wrinkles, but they are both good in most cases.) Our experiments also showed that cache investment for indexes significantly outperforms the distributed query processing techniques used by most existing commercial systems; sometimes by as much as a factor of ten.

Of course, there already has been a large body of related work about various aspects of caching: e.g., view or semantic caching [RCK+95, KB94, DFJ+96], caching for data warehouses [SSV96], cost models for caching [SW96], prefetching [PZ91, CKV93], etc. All previous work on caching, however, ignored the circular dependency between caching and query optimization. Using cache investment, we are the first to effectively break this circular dependency thereby allowing any kind of system to take the best advantage of caching. For example, with view caching, it is important to break this dependency in order to decide at which "level" of a plan to cache.

As stated in the introduction, we expect this work to have significant impact on many of today's systems. One example is SAP R/3 for which cache investment could relieve application programmers and system administrators from the burden to decide "by hand" which tables and indexes to cache and which operations to carry out at the application level and which at the server. Other examples include all database systems that are based on pure query shipping (e.g., relational systems such as Oracle, DB2, etc.) or pure data shipping (e.g., OODBs such as O_2, etc.). Cache investment would also help "truly" distributed systems such as System R* [WDH+81] or Mariposa [SAL+96]. These systems provide the flexibility to execute query operators at every site (like hybrid shipping), but usually provide no support for caching. Instead of caching, these systems support replication which is more static and coarser-grained. Cache investment can be used in addition to replication, and the combination of these two complementary techniques is an important avenue for future work.

Acknowledgments We would like to thank Markos Zaharioudakis for providing us with his index-caching SHORE code.

References

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

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

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

[CN97] S. Chaudhuri and V. Narasayya. An efficient, cost-driven index selection tool for microsoft SQL server. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 146–155, Athens, Greece, August 1997.

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

[DHKK97] J. Doppelhammer, T. Höppler, A. Kemper, and D. Kossmann. Database performance in the real world: TPC-D and SAP R/3. In SIGMOD [SIG97], pages 123–134.

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

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

[FK97] M. Franklin and D. Kossmann. Cache investment strategies. Technical Report CS-TR-3803, University of Maryland, College Park, MD 20742, May 1997. Submitted for journal publication. `http://www.db.fmi.uni-passau.de/ kossmann`.

[Fra96] M. Franklin. *Client Data Caching: A Foundation for High Performance Object Database Systems*. Kluwer Academic Press, 1996.

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

[HS76] E. Horowitz and S. Sahni. *Fundamentals of Data Structures*. Computer Science Press, Rockville, MD, USA, 1976.

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

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

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

[RCK+95] N. Roussopoulos, C. Chen, S. Kelley, A. Delis, and Y. Papakonstantinou. The ADMS Project: Views r us. *IEEE Data Engeneering Bulletin*, 18(2):19–28, June 1995.

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

[SFT88] M. Schkolnick, S. Finkelstein, and P. Tiberio. Physical database design for relational databases. *ACM Trans. on Database Systems*, 13(1):91–128, March 1988.

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

[SIG97] *Proc. of the ACM SIGMOD Conf. on Management of Data*, Tucson, AZ, USA, May 1997.

[SSV96] P. Scheuermann, J. Shim, and R. Vingralek. Watchman: A data warehouse intelligent cache manager. In VLDB [VLD96], pages 51–62.

[SW96] M. Sinnwell and G. Weikum. A cost-model-based online method for distributed caching. In *Proc. IEEE Conf. on Data Engineering*, Birmingham, GB, 1996.

[VLD96] *Proc. of the Conf. on Very Large Data Bases (VLDB)*, Bombay, India, September 1996.

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

[Yao77] S. B. Yao. Approximating block accesses in database organizations. *Communications of the ACM*, 20(4):260–261, April 1977.

[ZC97] M. Zaharioudakis and M. Carey. Highly concurrent cache consistency for indices in client-server database systems. In SIGMOD [SIG97], pages 50–61.