Loading a Cache with Query Results

Laura M. Haas
IBM Almaden

Donald Kossmann
University of Passau

Ioana Ursu
IBM Almaden

Abstract

Data intensive applications today usually run in either a client-server or a middleware environment. In either case, they must efficiently handle both database queries, which process large numbers of data objects, and application logic, which involves fine-grained object accesses (e.g., method calls). We propose a wholistic approach to speeding up such applications: we load the cache of a system with relevant objects as a by-product of query processing. This can potentially improve the performance of the application, by eliminating the need to fault in objects. However, it can also increase the cost of queries by forcing them to handle more data, thus potentially reducing the performance of the application. In this paper, we examine both heuristic and cost-based strategies for deciding what to cache, and when to do so. We show how these strategies can be integrated into the query optimizer of an existing system, and how the caching architecture is affected. We present the results of experiments using the Garlic database middleware system; the experiments demonstrate the usefulness of loading a cache with query results and illustrate the tradeoffs between the cost-based and heuristic optimization methods.

1 Introduction

Data intensive applications today usually run in either a middleware or client-server environment. Examples of middleware systems include business application, e-commerce or database middleware systems, while CAD and CAE systems are typically client-server. In either case, they must efficiently handle both database queries, which process large numbers of data objects, and application logic, with its fine-grained object accesses (e.g., method calls). In both architectures, application logic and query processing may be co-resident, and take place on a processor other than that on which the data resides. It is increasingly likely that some or all of the data will be on remote and/or nontraditional data sources that are expensive to access, such as web sources or specialized application systems.

Sophisticated optimization techniques reduce query processing times in these environments, while caching is used to reduce the cost of the application logic by avoiding unnecessary requests to the data sources. Applications often ask queries to identify objects of interest and then manipulate the result objects. Though it is now possible to do chunks of application logic in the query processor, applications still do much of the work themselves. Some applications require user interaction; others desire greater portability and ease of installation (e.g., big business applications such as Baan IV, Peoplesoft 7.5, or SAP R/3). In traditional systems, query processing and caching decisions are made in isolation. While this provides acceptable performance for these systems, it is a disaster for applications using data from the Internet. This query-and-manipulate pattern means that traditional systems access the data twice: once while processing the query, and then again, on the first method call, to retrieve and cache the object. If data is on the Internet, this will be prohibitively expensive. In some cases, the data source may not even be able to look up individual objects; hence this extra round trip is impossible.

In this paper we propose to load the cache with relevant objects as a by-product of the execution of a query. With this technique it is possible to get orders of magnitude improvements for applications that involve both queries *and* methods over expensive-to-access data. However, a naive implementation can do more harm than good. An application today can manually cache query results by explicitly selecting all the data for the object in the query itself. However, this may increase the cost of queries dramatically by forcing them to handle more data. For complex queries this effect may be large enough to more than offset the benefit. Therefore, the decisions of what to cache and when during query execution to do so should be made by the query optimizer in a cost-based manner.

The remainder of this paper is organized as follows. In Section 2, we elaborate on the motivation for our work, and discuss the caching of objects in our environment. While loading a cache with query results is essential when data is expensive or difficult to access, our ap-

**Proceedings of the 25th VLDB Conference,
Edinburgh, Scotland, 1999.**

proach can also be used to speed up applications in traditional two- or three-tier architectures as described above. For ease of exposition we will talk about "middleware" as the site of query processing and caching in the following sections. In a two-tier system, these activities would take place in the client. Section 3 presents alternative ways to extend an optimizer to generate query execution plans that load a cache with query results. We describe two simple heuristics, as well as a more sophisticated cost-based approach. Section 4 discusses our implementation of caching in the Garlic database middleware system, and Section 5 contains the results of performance experiments that demonstrate the need to load a cache with query results and show the tradeoffs of the three alternative ways of extending the query processor. Section 6 discusses related work, and Section 7 concludes the paper.

2 Caching in Middleware

2.1 A Motivating Example

To see why loading the cache with query results is useful, consider this (generic) piece of application code:

foreach o, o_2, o_5 **in**
(select r.oid, s_2.oid, s_5.oid
from R r, S_1 s_1, S_2 s_2, ...
where ...)
{ ... o.method(o_2, o_5); ... }

The query in this example is used to select relevant objects from the database. After further analysis and/or user interaction, the method carries out operations on these objects. The query can be arbitrarily complex, involving joins, subqueries, aggregation, etc. The method will involve accesses to certain fields of the object o and possibly to other objects (o_2, o_5) as well. *r.oid* refers to the object identifier of an object of collection R; this identifier is used to invoke methods on the object and to access fields of the object. Such a code fragment could be found in many applications. For example, an inventory control program might select all products for which supplies were low (and their suppliers and existing orders). After calculating an amount to order (perhaps with user input), it might invoke a method to order the product.

In a traditional middleware system this code fragment is carried out as follows:

1. the query processor tries to find the best (i.e., lowest cost) plan to execute the query.

2. the query processor executes the query, retrieving the object ids requested.

3. an interpreter executes the method, using the object ids to retrieve any data needed. To speed up the execution of methods that repeatedly access the same objects, the interpreter uses caching. Requests to access objects already in the cache can be processed by the interpreter without accessing the underlying data source(s), and a request to access an object not found in the cache would result in *faulting in* that object.

The key observation is that query processing does not affect caching in traditional systems: if the relevant objects of R are not cached prior to the execution of the query, these objects will not be cached as a by-product of query execution and they will have to be faulted in at the beginning of each method invocation. In an environment in which data access is slow, this can be extremely expensive – just as expensive, in fact, as processing the query. Loading the cache with query results avoids this extra cost of faulting in objects by copying the R objects into the cache while the query is executed; that is, it seizes the opportunity to copy the R objects into the cache at a moment at which the objects must be accessed and processed to execute the query anyway.

2.2 Caching Objects

Our goal is to decrease the overall execution time of applications, such as those described above, that use queries to identify the objects on which they will operate (i.e., on which they will invoke methods). There are many possible ways to accomplish this goal. In this paper, we focus on speeding up method execution, by essentially "pre-caching" the objects that methods will need. This pre-caching is possible in our environment, first, because, in executing the query, the query processor has to touch the needed objects anyway, and second, because in the architectures we consider, some portion of the query processing is done at the same site as that at which the methods are executed. Hence, the query processor has the opportunity to copy appropriate objects into a cache, for the methods to use.

Obviously, it will only be beneficial to cache objects that are subsequently accessed by the application program. Ideally, one would carry out a data flow analysis of the application program [ASU89] in order to determine which objects of the query result are potentially accessed. Unfortunately, such data flow analyses are impossible in many cases due to the separation of application logic and query processing – and interactive applications are totally unpredictable. Thus some heuristic approach to identifying the relevant objects is needed. It is likely that the objects whose oids are returned as part of the query result (i.e., objects of collections whose oid columns are part of the query's SELECT clause) are going to be accessed by the application program subsequently (why else would the query ask for oids?)[1]. We refer to such collections as *candidate collections*; these collections are candidates because objects of these collections are likely to be accessed. However, they are not guaranteed to be cached, as it might nevertheless not be cost-effective.

[1] Alternatively, we could assume that the application programmer gives hints that indicate which collections should be cached.

In the applications we are considering, queries and methods run in the same transaction. Hence we are only interested in intra-transaction caching in this paper, and cache consistency is not an issue. Our approach is particularly attractive in environments that do not support inter-transaction caching because transactions start with no relevant objects in the cache. Issues of locking and concurrency control are orthogonal to loading a cache with query results. In a middleware environment, it is often undesirable or impossible to lock data in the sources for the duration of the transaction. Under such circumstances, our approach may cause an application to produce different output; but, in some sense, this output can be seen as better output because our approach guarantees that the methods see the same state of an object as the query.

In this paper, we assume that the granularity of caching is an entire object. (We discuss how an object is defined in Section 4.) To cache the object, the whole object must be present. One may argue that we should only copy those fields of objects that are retrieved as part of the query anyway. However, state-of-the-art caches cache in the granularity of whole objects (e.g., the cache of SAP R/3 [KKM98]). This is necessary for pointer swizzling [Mos92, KK95] and to organize the cache efficiently (i.e., avoid a per attribute overhead in the cache). One may also argue that the granularity of caching and data transfer should be a group of objects or a *page*; caching and data transfer in such a granularity, however, is not possible in systems like Garlic.

A consequence of this approach is that caching during query execution is not free. It introduces additional cost, as no attribute of an object may be projected out before the object is cached, even if the attribute is not needed to compute the query result. This has two implications. First, objects should only be cached if the expected benefit (overall application speedup due to faster execution of methods) outweighs the cost in query execution time. Second, the point in the query execution at which objects are cached will affect the cost and benefit. If caching occurs too early, irrelevant objects may be cached, and might even flood the cache, squeezing out more relevant objects. If caching occurs too late, the intermediate results of query processing will be larger due to the need to preserve whole objects for caching. Consider, for instance, a query that involves a join between R and S and asks for the *oid* of the R objects that qualify: joining only the *oid* column of R with S is cheaper than joining the *whole* R (i.e., *oid* and all other columns) with S, especially if the *oid* column of R fits into main memory and the *whole* R does not. Because caching impacts the size of intermediate results, it should also impact join ordering; for instance, joins that filter out many objects of candidate collections should perhaps be carried out early in a query plan if caching is enabled. Hence, the best way of executing the query may be different depending on whether we are caching objects.

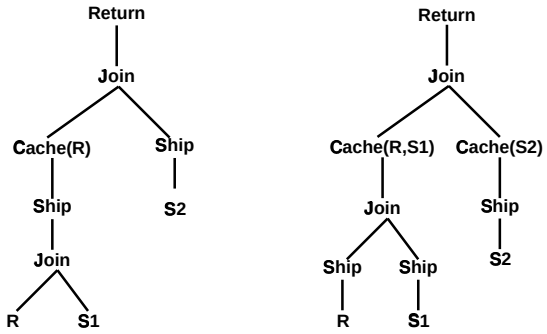


Figure 1: Example Cache Enhanced Query Plans

In summary, our goal is to speed up the execution of methods by caching the objects they need (as indicated by the select list of a query) during execution of that query. The granularity of the cache is an object, and caching objects during query execution incurs costs that can affect the choice of query execution plan. As a result, we will allow the query optimizer to decide what objects to cache, and when.

3 Caching During Queries

In this section, we describe ways to extend the query processor of a middleware system in order to generate plans which cache relevant objects. We introduce a new *Cache* operator which the query optimizer can use to indicate where in a plan objects of a particular collection should be cached. A *Cache* operator copies objects from its input stream into the cache and projects out columns of the input stream which are not needed to produce the query results. A *Cache* operator takes two parameters, one that specifies which objects of the input stream should be copied into the cache, and one that specifies which columns should be "passed through" to the next operator (not projected out). The plans shown in Figure 1 could be produced by the enhanced query optimizer. The *Cache* operator of the first plan copies objects of collection R; the first *Cache* operator of the second plan copies objects of R and S_1 while the second copies S_2 objects. A plan may contain several *Cache* operators if the objects of more than one collection are to be cached; however, it makes no sense to have two *Cache* operators for the same collection in a plan. The *Return* operators pass the query results to the application program. The *Ship* operators pass intermediate query results from a data source to the middleware; since *Cache* operators are always executed by the middleware, all *Cache* operators must be placed somewhere above a *Ship* operator and below the final *Return* operator.

In order to generate such plans, the query optimizer must decide (1) for which collections involved in a query to include *Cache* operators in a query plan, and (2) where to place *Cache* operators in a query plan. We present three approaches. The first two are heuristics which serve as baselines for our study. The third approach is cost-

based cache operator placement: this approach is likely to make better decisions (i.e., produce better plans), but increases the cost of query optimization.

3.1 Cache Operators at the Top of Query Plans

The first approach makes the two cache operator placement decisions in the following heuristic way: (1) generate a *Cache* operator for every candidate collection, and (2) place all *Cache* operators high in a query plan. This approach corresponds to what an application could do manually, and is based on the principle that *all* relevant objects (objects which are part of the query result and belong to candidate collections) should be cached during the query and *no* irrelevant objects (those not part of the query result) should be cached. In detail, this approach works as follows:

1. rewrite the `SELECT` clause of a query, replacing all occurances of `oid` by `*`.

2. optimize the rewritten query in the conventional way.

3. include *Cache* operators for the collections whose *oid* columns are requested in the `SELECT` clause of the query, and place those *Cache* operators at the top of the query plan generated in Step 2 (i.e., just below the *Return* operator); remember that *Cache* operators carry out projections so that the right columns for the original query are returned.

4. push down *Cache* operators through *non-reductive* operators. A non-reductive operator is an operator that does not filter out any objects. Examples are *Sort* operators and certain functional joins for which integrity constraints guarantee that all objects satisfy the join predicate(s) (see [CK97] for a formal definition of non-reductive operators).

The push-down of *Cache* operators through non-reductive operators (Step 4) reduces the cost of executing the query and at the same time obeys the principle that only relevant objects are copied into the cache. Suppose, as an example, that a *Cache* operator is pushed below a *Sort*: the cost of the *Sort* is reduced because the *Sort* operator works on *thin* tuples, because the *Cache* operators project out all the columns that were added as part of the rewriting in Step 1. At the same time, no irrelevant objects are copied into the cache because the *Sort* does not filter out any objects.

While pushing down *Cache* operators through non-reductive operators is certainly an improvement, this "caching at the top" approach clearly does not always produce good cache-enhanced plans. Because *Cache* operators impact the size of intermediate results, the placement of *Cache* operators should also impact join ordering; however, the heuristic ignores this interdependency. Furthermore, *Cache* operators high in a plan force lower operators to handle *thick* tuples with high extra cost. The heuristic basically assumes that the extra cost incurred by plans with *Cache* operators is always outweighed by the benefit of these *Cache* operators for (future) method invocations – an assumption which is not always valid, even when data accesses are expensive (Section 5.4).

3.2 Cache Operators at the Bottom of Query Plans

The second approach, "caching at the bottom", makes the following cache operator placement decisions: (1) generate a *Cache* operator for every candidate collection, and (2) place all *Cache* operators low in a query plan. Like the "caching at the top" heuristic, the "caching at the bottom" heuristic assumes that the benefits of *Cache* operators for candidate collections always outweigh the cost incurred by the presence of *Cache* operators. However, the "caching at the bottom" heuristic places *Cache* operators low in query plans, following the principle that columns which are only needed for caching and not to evaluate the query itself should be projected out as early as possible. Thus, the "caching at the bottom" approach affects the cost of other query operators (i.e., joins, group bys, etc.) as little as possible, but it might copy objects into the cache that are not part of the query result and which would be filtered by these other query operators.

In detail, the "caching at the bottom" approach works as follows:

1. optimize the original query in the conventional way.

2. for each leaf node of the resulting plan, if the operator accesses a candidate collection, expand the list of attributes returned to include all the attributes of the objects.

3. place a *Cache* operator for that collection above each such leaf operator.

4. pull up *Cache* operators that sit below pipelining operators (e.g., filters or nested-loop joins).

Cache operator pull-up in the "caching at the bottom" approach is analogous to *Cache* operator push-down in the "caching at the top" approach. Push-down heuristics reduce the cost of a query without increasing the number of *false cache insertions* (adding objects to the cache that do not participate in the query result, hence will not be used later). Pull-up heuristics reduce the number of *false cache insertions* without increasing the cost of a query. Consider as an example a *Cache* operator that sits below a pipeline operator which filters out some of its input tuples. Moving the *Cache* operator above that pipeline operator will reduce the number of objects copied into the cache, without increasing the cost of the pipeline operator because the cost of a pipeline operator does not depend on the width of the tuples it processes.

3.3 Cost-based Cache Operator Placement

It should be clear from the previous two subsections that there is a fundamental tradeoff between "high" and "low" *Cache* operators: the higher a *Cache* operator, the lower the number of *false cache insertions*, and the higher the number of other query operators that sit below the *Cache* operator and operate at increased cost because they must process *thick* tuples. The "caching at the top" and "caching at the bottom" heuristics attack this tradeoff in simple ways; obviously, there are situations in which either one or even both approaches do not find the best place to position a *Cache* operator in a query plan.

In this section, we show how a query processor can make *Cache* operator placement decisions in a cost-based manner. The approach is based on the following extensions:

1. extend the enumerator to enumerate alternative plans with *Cache* operators
2. estimate the cost and potential benefit of *Cache* operators to determine the best plan; the cost models for other query operators (e.g., joins, etc.) need not be changed
3. extend the pruning condition of the optimizer to eliminate sub-optimal plans as early as possible

We describe these three extensions in more detail in the following subsections.

3.3.1 Enumeration of Plans with *Cache* Operators

The implementation of the cost-based placement strategy is integrated with the planning phase of the optimizer. We discuss the necessary changes in the context of a bottom-up dynamic programming optimizer [SAC+79]. Optimizers of this sort generate query plans in three phases. In the first phase, they generate plans for single collection accesses. In the next phase, they generate plans for joins. They first enumerate the two-way joins, using the plans built in the first phase as input. Likewise, they then plan three-way joins, using the plans previously built (for single collections and two-way joins), and so on, until a plan for the entire join is generated. The final phase then completes the plan by adding operators for aggregation, ordering, unions, etc. Each plan has a set of *plan properties* that track what work has been done by that plan. In particular, they record what collections have been accessed, what predicates applied, and what attributes are available, as well as an estimated cost and cardinality for the plan[2]. Each operator added to a plan modifies the properties of that plan to record what it has done. At the end of each round of joins, as well as at the end of each phase, the optimizer *prunes* the set of generated plans, finding plans which have done the same work (have the same properties) and eliminating all but the cheapest.

[2] There are several other properties that are tracked; we only list the most relevant for this paper.

Plan 1: Index Scan - $A_{\mathtt{thick}}$
Plan 2: Index Scan - $A_{\mathtt{thin}}$
Plan 3: Relation Scan - $A_{\mathtt{thick}}$
Plan 4: Relation Scan - $A_{\mathtt{thin}}$
Plan 5: Cache(A) - Ship - Index Scan - $A_{\mathtt{thick}}$
Plan 6: Cache(A) - Ship - Relation Scan - $A_{\mathtt{thick}}$

Figure 2: Plans for Accessing Table A

Only a few changes need to be made to an existing optimizer to allow it to generate plans with *Cache* operators. First, we have to define what a *Cache* operator does to a plan's properties. *Cache* projects out (i.e., does not pass on to higher operators) unneeded attributes, so it changes the attribute property. It also will affect the cost, as discussed in Section 3.3.2 below. Next, the first and second phases must be modified to generate alternative plans with *Cache* operators. In modern dynamic programming optimizers [Loh88, HKWY97], this corresponds to adding one *rule* to each of those phases. In the access phase, in addition to the normal (*thin*) plans for a collection, which select out just the attributes needed for the query, the new rule will also generate plans for getting all the attributes of the objects in the collection (*thick plans*), if the collection is one of those whose *oid* column is selected by the query (i.e., a candidate collection). In addition, the rule will generate extra plans which consist of a *Cache* (and *Ship*) operator above each of the thick plans. Figure 2 shows the six plans that would be generated in phase one of enumeration if the collection access could be done by either scanning the collection or by scanning an index. If *thick* and *thin* coincide (i.e., all columns of A are needed to produce the query result, regardless of caching), only four plans would be enumerated, as Plans 1 and 3 would be identical to 2 and 4, respectively.

Similarly, in the join planning phase, the enumerator must consider possible caching plans in addition to normal join plans. Since there will be a thick plan for each candidate collection, we will automatically get joins with thick result objects. On top of these, we add appropriate *Cache* operators during each round of joining. We can consider caching any subset of *available* candidate collections in a given plan, where *available* means that the plan's properties indicate that that collection has been accessed, that no other *Cache* operator for that collection is present in the plan, and that the full objects are present (it's a thick plan for that collection). This, of course, can cause an exponential explosion in the number of plans that must be considered. For example, Figure 3 shows four basic join plans and five caching plans for a two table join query; actually, even more plans are possible taking into account that more than one join method is applicable and that *Ship* operators can be placed before or after the joins. In Section 3.3.3, we discuss how ag-

Plan 1: Join - Ship - Scan - A_{thick}
- Ship - Scan -B_{thick}
Plan 2: Join - Ship - Scan - A_{thin}
- Ship - Scan -B_{thick}
Plan 3: Join - Ship - Scan - A_{thick}
- Ship - Scan -B_{thin}
Plan 4: Join - Ship - Scan - A_{thin}
- Ship - Scan -B_{thin}
Plan 5: Cache(A) - Join - Ship - Scan - A_{thick}
- Ship - Scan -B_{thick}
Plan 6: Cache(B) - Join - Ship - Scan - A_{thick}
- Ship - Scan -B_{thick}
Plan 7: Cache(A,B) - Join - Ship - Scan - A_{thick}
- Ship - Scan -B_{thick}
Plan 8: Cache(B) - Join - Ship - Scan - A_{thin}
- Ship - Scan -B_{thick}
Plan 9: Cache(A) - Join - Ship - Scan - A_{thick}
- Ship - Scan -B_{thin}

Figure 3: Plans Generated for $A \bowtie B$
A, B are candidate collections

gressive pruning can help control this explosion.

3.3.2 Cost/Benefit Calculation of *Cache* Operators

Since *Cache* operators can only be applied on whole objects, their presence increases the cost of underlying operators (because these underlying operators must work on more data). Further, since *Cache* operators project out the columns not needed for the query result, their properties (other than cost) are the same as a simple (non-caching) thin plan. For example, Plans 2, 4, 5 and 6 in Figure 2 have the same properties, excluding cost. Plans with *Cache* operators have done more work to get to the same point; they can survive, therefore, only if the *Cache* operators have a negative cost. At the beginning of optimization, a potential *benefit* is computed for each collection to be cached. The *cost* of a *Cache* operator is defined as the *actual cost* to materialize its input stream *minus* the estimated *benefit*, or savings, from not faulting in objects in future method invocations. The actual cost of the *Cache* operator is proportional to the cardinality of the input plan, and represents the time to copy objects into the cache, and do the project to form the output stream.

The benefit is considerably trickier to estimate. Fortunately, a reasonably detailed model is possible, and is sufficient for choosing good plans. To compute the benefit of a collection, we need to know how many distinct objects of the collection will be part of the query result. For simplicity, we will refer to this number as the *output* of the collection for this query. We assume that the application will invoke methods on a certain fraction F (e.g. 80 %) of the objects in the query result. The benefit, B, is proportional to the output, O: $B = k \times F \times O$, where k represents the time to fault in the object[3]. k, F, and the output of a collection are constant for a given query; they do not depend on the plan for the query, or when (or if) caching occurs. Thus, the benefit can be computed before planning begins. For complete accuracy, B should include a factor f_1 representing the fraction of the relevant objects not already in the cache; however, the overhead to estimate f_1 is not justified given the accuracy we can achieve for other parts of the formula, so we ignore this factor and assign F a lower value accordingly.

The tricky part is how to estimate the output. One approach is to let the optimizer do it. For this alternative, to find the output of a collection R, the optimizer is asked to plan a modified version of the original query, such that the original select list is replaced by "distinct R.oid". The result cardinality of this query is the required output. Note that since the plan for this modified query is unimportant, the optimizer can use any greedy or heuristic approach it wants to reduce optimization time, as long as it does use its cardinality estimation formulas. However, this approach is still likely to be expensive, especially for large queries in which multiple collections are candidates for caching, as the optimizer will be called once per candidate collection, and then again to plan the actual query. Nor is the result guaranteed to be accurate; it will be only as good as the optimizer's cardinality estimates.

Instead, we devised a simple algorithm for estimating output [HKU99]. This approach has much less overhead and estimates the output of a collection with accuracy close to that of the optimizer for queries where the join predicates are independent. The algorithm takes a query as input, and returns an estimate of the output of each candidate collection for the query. The algorithm essentially emulates the optimizer's cardinality computations, but without building plans. It starts by estimating the effect of applying local predicates to the base collections, using the optimizer formulas. It then heuristically chooses an inner for each join and "applies" the join predicate to the inner's output. The output of a collection is taken to be the minimum value among its initial cardinality, its output after applying the most selective local predicate (if any) and its output after applying the most selective join predicate (if any). The algorithm seems to provide a good compromise between accuracy and overhead, though it needs tuning for joins over composite keys.

3.3.3 Pruning of Plans with *Cache* Operators

At the end of each phase of planning, and at the end of each round of joins, the optimizer examines the plans that have been generated, and "prunes" (i.e., throws away) those which are at least as expensive as some other plan that has equivalent or more general properties. Thin plans are less general (because they make available fewer attributes) than thick ones; hence, although thick plans

[3] k depends on the data source and object. [ROH98] describes how an optimizer can assess the value of this parameter.

are typically more expensive, they will not naturally be pruned in favor of thin plans.

This is good, in terms of ensuring that all possible caching plans are examined. However, as described in Section 3.3.1, it also leads to an exponential explosion in the number of plans. Fortunately, since the *Cache* operator only passes through those attributes needed for the query, it creates thin plans (or at least, thinner plans) that compete with each other. For example, in Figure 2, of the six plans generated for accessing collection A in the first phase of optimization, at most two will survive: one thick plan and one thin plan (if it is cheaper than the thick one). The thin survivor could either be a caching plan (e.g., Plan 6) or an original thin plan (e.g., Plan 2). In the join phase, the maximum number of plans that survives each round is 2^n, where n is the number of candidate collections in this round. So in Figure 3, four plans could survive: one in which both A and B are thick, one in which both are thin, one in which A is thick and B thin, and one in which B is thick and A thin (for example, the survivors might be Plans 1, 2, 6 and 7).

However, under certain conditions we can safely prune a thick plan in favor of a thin – and the sooner we eliminate such plans the better for optimization times. In particular, we can prune the thick plan for a candidate collection A if:

$$Cost_{A_{thin}} \leq Cost_{A_{thick}} + Cost_{A_{CacheBest}} - Benefit$$

where $Cost_{A_{CacheBest}}$ is the minimum actual cost incurred to cache a collection and corresponds to the case where the *Cache* operator sits directly above that join that results in the minimum number of output tuples from the collection. It can be computed before optimization, during the output calculations described in Section 3.3.2. The condition basically says that if we assume the minimal possible cost for caching A (lowest actual cost less constant benefit), and that is still more than the cost of a thin plan for A, then there is no point in keeping the thick plan, as caching A is not a good idea.

3.4 Other Strategies and Variants

In this section, we presented three alternative ways to generate plans with *Cache* operators. These three approaches mark cornerstones in the space of possible strategies for integrating *Cache* operator placement into a query processor. The first two approaches are simple strategies that always place *Cache* operators either at the top or at the bottom of query plans. Neither approach causes much overhead during query optimization, but they are likely to make sub-optimal decisions in many cases. The third approach is a full-fledged, cost-based approach for determining cache operator placement. This approach can be the cause of significant additional overhead during query optimization, but is likely to make good decisions.

We can imagine many approaches that make better decisions than the "caching at the top" and "caching at the bottom" heuristics at the expense of additional overhead, or approaches that are cheaper than "cost-based caching" at the risk of making poor decisions in some cases. We describe here just a few variants:

cost-based Cache operator pushdown: rather than push *Cache* operators down through *non-reductive* query operators only, this variant would push a *Cache* operator down through another operator if the result would be a lower cost plan, using the cost model and cost/benefit calculations for *Cache* operators of Section 3.3.2.

cost-based Cache operator pull-up: *Cache* operator pull-up can also be carried out during post-processing of plans in a cost-based manner, instead of pulling *Cache* operators up only through pipeline operators.

flood-sensitive Cache operator elimination: The "caching at the bottom" variant can be extended in such a way that *Cache* operators that would flood the cache because they are applied to too many objects (according to the cardinality estimates of the optimizer) are eliminated from the plan.

rigorous pruning in cost-based approach: There are several possible variants of the "cost-based caching" approach which more aggressively prune plans, even when it may not be wholly "safe" to do so (in other words, they may discard plans that could be the basis of winning plans later on). These variants reduce the cost of query optimization considerably, at the expense of perhaps missing good plans. For example, one aggressive variant might generalize the pruning condition of Section 3.3.3, and always keep at most one of the alternative plans at the end of the round. A somewhat gentler variant might keep two plans at the end of each round of plan generation: a "pure" thick plan, that is, a plan in which all attributes of all candidate collections of the plan are present, and a "pure" thin plan, that is, a plan in which no attributes not necessary for the original query are present.

4 Implementation Details

We implemented all three cache operator placement strategies described in the previous section and integrated them into the Garlic database middleware system. In this section, we describe the major design choices we made in our implementation.

4.1 Double Caching Architecture

Figure 4 shows the overall design of the cache manager and query execution engine. Our implementation involves a double caching scheme. There is a *primary cache* used by the application, while *Cache* operators load objects into a *secondary cache* during query execution. From the secondary cache these objects are copied into the primary cache when they are first accessed by a method. *Resident object tables* (ROT) in both the primary and secondary cache are used to quickly find an

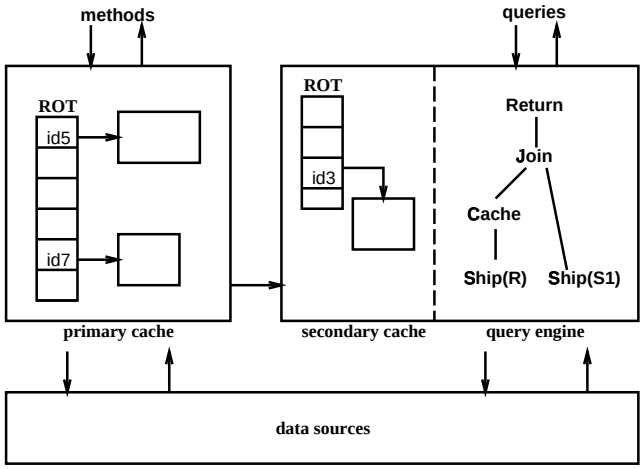


Figure 4: Double Caching Architecture

object in the cache. *Cache* operators only copy objects into the secondary cache that are not present in either the primary or the secondary cache. Thus, they waste as little main memory for double caching as possible and avoid copying objects into the secondary cache multiple times if the input stream of the *Cache* operator contains duplicates. During method invocations, an object is faulted into the primary cache from the data sources if it is not found in the primary or the secondary cache, just as in a traditional middleware system.

The double caching scheme shown in Figure 4 has two important advantages. First, copying objects into a secondary cache, rather than directly into the primary cache, prevents the primary cache from being flooded with query results, thus displacing frequently used objects. Consider, for example, a case in which the query optimizer estimates that the *Cache* operator copies, say, 100 objects; but in fact, the optimizer errs because of outdated statistics and the *Cache* operator would in fact copy millions of objects into the cache. The double caching scheme makes it possible to control and limit the impact of *Cache* operators. Second, the overhead of copying objects into the cache as a by-product of query execution can be reduced in such a double caching scheme. In the primary cache, objects are managed and replaced in the granularity of objects—this is reasonable because individual objects are faulted in and replaced in the primary cache during method invocations. The secondary cache, on the other hand, is organized in chunks; that is, when a *Cache* operator begins execution it will allocate space for, say, 1000 objects in the secondary cache, knowing that it is likely to copy many objects. In other words, the double caching scheme makes it possible to efficiently *bulkload* the cache with relevant objects.

However, the double caching scheme also has some disadvantages: (1) it incurs additional computational overhead in order to copy objects from the secondary cache into the primary cache when the objects are needed; (2) it does waste main memory because after an object has been copied from the secondary into the primary cache, it is cached twice; (3) it requires some (albeit little) tuning effort—this is the flip side of the coin which provides better control over the impact of *Cache* operators. In our experience, the advantages of the double caching scheme outweigh these disadvantages, but, in general, the tradeoffs strongly depend on the kind of application being processed by the middleware system.

4.2 Caching in Middleware for Diverse Sources

Garlic has been designed with an emphasis on handling diverse sources of information, especially sources that do not have traditional database capabilities, though they may offer interesting search and data manipulation capabilities of their own. Loading the middleware cache with query results is particularly attractive for systems like Garlic. First, communicating with some sources may be expensive in Garlic; almost any Web source, for example, will have a highly variable and typically long response time. In such situations, the benefit of *Cache* operators is particularly high (i.e., parameter k is large). Second, some sources are unable to just produce an object given its oid; that is, they do not support the faulting in of objects. Applications that operate on data stored in such data sources *must* load relevant objects as a by-product of query execution; otherwise, such applications simply cannot be executed.[4]

Loading the middleware cache with query results also raises several challenges in this environment. Diverse sources have diverse data. It may not always be practical to cache an entire object. For example, an object may have large and awkward attributes that should only be brought to the middleware if they are really needed. Alternatively, it may be desirable to cache values that are actually computed by methods of a data source because these values are frequently referenced by application programs. So, a flexible notion of "object" is needed. Garlic provides some flexibility in defining objects. Garlic communicates with sources by way of wrappers [RS97]. A wrapper writer must understand the data of a source and describe it in terms of objects. The description can indicate for each attribute (and method) of an object whether it should be part of the cached representation of the object. Garlic has access to this description during query processing, and can use it to decide what attributes and/or methods to include in a thick plan. Ideally, however, we would cache *application objects* which could include data from several collections, possibly from different data sources, and let programmers define such *application objects* for each application program individually. At present we have no mechanism to cache such user-defined application objects, but caching the underlying objects serves the same purpose, by bringing the data needed to construct the application

[4] In such situations, our cost-based approach must be extended to make sure that the winning plan contains a *Cache* operator.

Collection	Base cardinality	Data source
course	12,000	UDB
department	250	UDB
coursesection	50,000	UDB
professor	25,000	UDB
student	50,000	UDB
kids	116,759	UDB
NotesCourses	12,000	Notes
NotesDepartments	250	Notes
WWWPeople	25,000	WWW

Table 1: Test Data Sources and Object Collections

Query	Data sources	Output cardinality
select c.oid from course c where c.deptno < 11	UDB	500
select c.oid from NotesCourses c where c.course_dept < 11	Notes	500
select p.oid from WWWPeople where p.WWWcategory = 'professor' and p.WWWname like 'professorName15%'	WWW	500

Table 2: Benchmark Queries for Experiment 1

object to the middleware server.

5 Experiments and Results

This section presents the results of experiments that demonstrate the utility (and even, the necessity) of loading a cache with query results by studying the overall running times of applications that involve queries and methods. Next, we look at how query planning time is affected by the three *Cache* operator placement strategies. Finally, we compare the quality of plans produced by the three approaches. We begin with a description of the experimental environment.

5.1 Experimental Environment

The experiments were carried out in the context of the Garlic project, using the double caching architecture described in Section 4.1. For our experiments, we adapted the relational schema and data from the BUCKY benchmark [CDN+97] to a scenario suitable for a federated system. The test data is distributed among three data sources: an IBM DB2 Universal Database (UDB), a Lotus Notes version 4.5 database, and a World Wide Web (WWW) source. The WWW source is populated with data from UDB at the time of query execution using IBM's Net.Data product. The data collections, base cardinalities, and distribution among data sources are shown in Table 1. The Garlic middleware and the UDB and WWW databases run on separate IBM RS/6000 workstations under AIX; the Notes database resides on a PC running Windows NT. All machines are connected by Ethernet. In all experiments, the middleware cache is initially empty.

5.2 Experiment 1: The Value of Caching

The first set of experiments shows the importance of caching in general, and of our *enhanced caching* (loading the cache with query results) in particular. We measured the running times of three simple application programs that initiate the execution of a query and invoke two methods on each object of the query result. The queries used in the three application programs are given in Table 2; they are simple one-table queries against the UDB, Lotus Notes, and WWW databases. For these simple queries, all three *Cache* operator placement strategies presented in Section 3 produce the same plan: *Cache-Ship-Scan*. Each method involves reading the value of one attribute of the object to which the method is bound. The size of the primary and secondary cache are chosen such that all relevant objects fit in both. We ran each application program ten times (beginning with an empty cache each time) and report on the average running times.

	UDB	Notes	WWW
no caching	47.8	22.9	3538.5
traditional caching	22.9	18.2	1762.3
enhanced caching	2.2	12.7	11.9

Table 3: Total Running Time [secs]

Table 3 shows the results. As expected, *enhanced caching* wins in all cases. The gains are particularly pronounced for the WWW application because interaction with the WWW database, as required to fault in objects, is particularly expensive—even if the WWW server is only lightly loaded and has all information available in main memory. The savings in cost are relatively low for the Notes application because faulting in objects from the Notes database is quite cheap so that the cost of query processing dominates the overall cost of the application in this case. In all cases, *traditional caching*, which faults in objects when they are used for the first time as part of a method invocation, beats *no caching* because it saves the cost of interacting with the data sources for the second method invocation.

In this experiment, the application program accesses *all* objects returned by the query; i.e., $F = 1$. For smaller F, the savings obtained by traditional and enhanced caching are less pronounced. As mentioned in Section 3.3.2, the benefit increases linearly with F; in the extreme case, for $F = 0$, no caching and traditional caching have the same running time as enhanced caching (in fact, a little better).

5.3 Experiment 2: Query Planning Times

The next experiment studied the planning times of the three *Cache* operator placement strategies. The two parameters that impact the planning time most are the number of collections involved in the query and the number of candidate collections. Our queries join collections stored in UDB and Notes. We varied the number of collections involved in the query and in all cases, all collections were considered candidate collections. Thus, these queries can be seen as tough cases which are expensive to optimize.

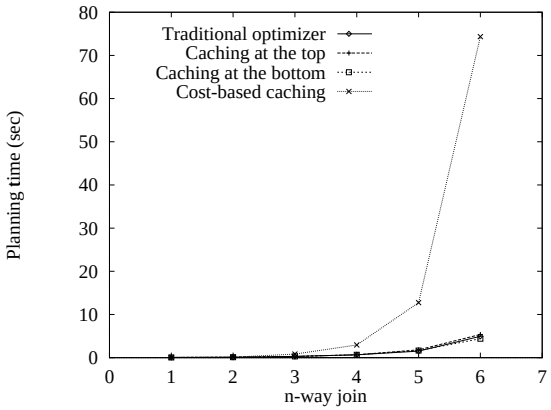


Figure 5: Planning Times for UDB/Notes Queries

Figure 5 shows the resulting planning times for each of the three approaches presented in Section 3. As a baseline, we also show the running time of a traditional optimizer that does not generate plans with *Cache* operators. Again, there are no surprises. The full-fledged cost-based approach becomes prohibitively expensive if there are more than four candidate collections in a query. At this point one of the two heuristics or the variants proposed in Section 3.4 should be used. Up to that point, however, the cost-based approach has negligible overhead and can safely be used. Comparing the "caching at the bottom," "caching at the top," and "traditional optimizer" lines, we see that the two heuristics have virtually no overhead.

5.4 Experiment 3: The Right Caching Decisions

The last set of experiments demonstrates the need to carry out cost-based *Cache* operator placement in certain situations. The experiments show: 1) how a *Cache* operator at the top can increase the cost of the other operators that sit below; 2) the overhead introduced by unnecessarily caching a large number of objects when a *Cache* operator is placed at the bottom; 3) the need to avoid flooding the secondary cache with irrelevant objects; and 4) that it is not always beneficial to have *Cache* operators for all candidate collections, even when accessing slow sources. We used queries over collections from the UDB and WWW databases. The queries and the best execution plan for each query are presented in Figure 6. "Caching at the top" works best for the first query; for the second query, "caching at the bottom" works best; and for the third query, no *Cache* operator at all should be generated. We again measured the total execution time of three simple application programs that each execute one of these queries and invoke one method on each object returned by that query. The method simply reads the value of one attribute. The size of the primary cache was set to 1000 objects which is more than enough to hold all objects involved during method invocations. For the first query (Q1), we studied two configurations for the secondary cache: (a) *medium*, with a capacity of 1000 objects, and (b) *large*, with a capacity of 6000 objects. We varied the size of the secondary cache for Q1 in order to study the implications of loading the cache with irrelevant objects, in particular for the "caching at the bottom" approach. For the other two queries, a *medium* secondary cache was sufficient in all cases, so we only show the results obtained using such a *medium* secondary cache.

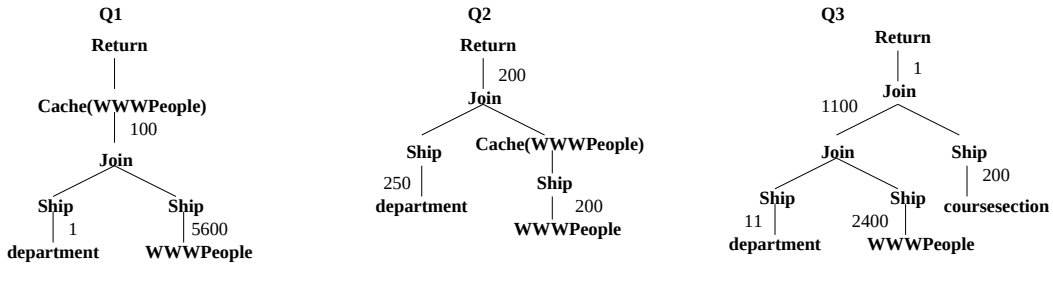
	Q1(large)	Q1(med)	Q2	Q3
no caching	405.5	405.5	842.5	129.2
traditional caching	405.5	405.5	842.7	129.9
caching at the top	71.3	71.3	49.8	177.5
caching at the bottom	76.0	415.8	34.9	141.9
cost-based caching	71.4	71.4	35.1	130.7

Table 4: Total Running Time [secs]
size of sec. cache: medium=1000 obj.; large=6000 obj.

Table 4 shows the results. We can see that the cost-based approach to loading the cache with query results shows the overall best performance, making the right caching decisions in all situations. The "caching at the top" approach, as expected, makes suboptimal decisions for Q2 and Q3, and the "caching at the bottom" approach makes suboptimal decisions for Q1 and Q3. The "caching at the bottom approach" shows particularly poor performance if it floods the secondary cache, so that few relevant objects are loaded as a by-product of executing the query (Q1 with a medium-sized secondary cache). "Caching at the bottom" is never much worse than traditional caching or no caching at all, and it can, therefore, be seen as a *conservative* method of extending today's database systems to load a cache with query results. The "caching at the top" heuristic, on the other hand, is as much as 37% more expensive than traditional caching in our experiments, and could easily be more. In these experiments, traditional caching and no caching show approximately the same performance because every result object is accessed exactly once as part of the method invocations.

6 Related Work

Most work on data processing in distributed systems has focused either on query processing or on caching, and most middleware systems today are built in such a way that query processing does not affect caching and vice versa. For example, SAP R/3 [BEG96, KKM98] is a very popular business administration system that supports the execution of (user and pre-defined) queries and methods, processing applications that involve both as described in Section 2.1. Persistence [KJA93] is a middleware system that enables the development of object-oriented (C++, Smalltalk, etc.) applications on top of a relational database system. That system typically pushes down the execution of queries to the relational database system and executes methods in the middleware using caching. Query processing and caching do not interact in




```
select p.oid
from WWWPeople p, department d
where p.WWWdno=d.dno and
      p.WWWcategory='staff' and
      p.WWWname like 'staffName1%' and
      d.name = 'deptname59'
```

```
select p.oid
from WWWPeople p, department d
where p.WWWdno=d.dno and
      p.WWWcategory='professor' and
      p.WWWdepartment like 'deptname25%'
      and d.dno<251
```

```
select p.oid
from WWWPeople p, department d, coursesection s
where p.WWWdno=d.dno and
      p.WWWid=s.instructorid and
      p.WWWcategory='professor' and
      p.WWWname like 'professorName2%' and
      d.name like 'deptname11%' and d.dno<120 and
      s.deptno=19
```

Figure 6: Benchmark Queries for Experiment 3

either system, so both would benefit from the techniques presented in this paper.

Database systems that have a *data shipping* architecture naturally load a cache with query results; examples are most object-oriented database systems such as O_2 [D+90]. These systems bring all the base data to the middleware (or client) to evaluate a query and that base data is then cached for subsequent queries and methods, if the cache is large enough. In some sense, data shipping, therefore, corresponds to the "caching at the bottom" approach – however, there is no *Cache* operator pull-up and no way to execute joins at data source(s). This causes data shipping to perform poorly for many types of queries [FJK96].

Another experimental database system that supports query processing and caching is KRISYS. In an early version which was targeted for engineering applications, KRYSIS used queries to load the cache with relevant objects [HMNR95], as proposed in our work. However, that version only supported a variant of the "caching at the top" approach (without *Cache* operator pushdown). In a more recent version [DHM+98], KRYSIS supports predicate-based caching. Predicate-based caching [KB94], like view caching [Rou91] and semantic caching [DFJ+96], makes it possible to cache the results of queries. The purpose of predicate-based caching, however, is to use the cache in order to answer future queries (rather than for methods). Hence, it requires significantly more complex mechanisms for tracking cache contents, and is not geared for the lookup of individual objects.

Two further lines of work are relevant. The first is cache investment [FK97]. Cache investment also extends a query processor to make it cache-aware. Again, however, the purpose of cache investment is to load the cache of the middleware in such a way that future queries (rather than methods) can be executed efficiently. The second related line of work is prefetching [PZ91, CKV93, GK94]. The purpose of prefetching is to bring objects into the cache before they are actually accessed. Prefetching, however, is carried out as a separate process, independent of query processing.

7 Conclusion

In this paper, we showed that caching objects during query execution dramatically speeds up applications that involve both queries and methods in a middleware (or client server) environment. The performance wins that can be achieved by this method are huge; they are particularly high in environments in which interactions with the data sources are very expensive; e.g., data sources on the Internet. In certain scenarios, loading a cache with query results in this way is even necessary; such a situation arises in heterogeneous database environments in which some data sources are not able to respond to requests for individual objects.

To implement our approach we extended the cache manager and the query processor of a middleware system. We used a double caching scheme to reduce the overhead of our approach and to avoid flooding the primary cache with (useless) objects as a by-product of query execution. We explored three alternative ways of extending the query processor: "caching at the top," "caching at the bottom," and "cost-based caching." The first two approaches are simple heuristics which can be easily incorporated in an existing query processor and which typically do not increase query optimization times; however, the "caching at the top" approach can result in substantially increased query execution times, while the "caching at the bottom" approach may cache many useless objects, thereby causing additional overhead and providing no benefit if the cache is too small. The third approach is significantly more complex to implement and increases optimization times of complex queries substantially, but is always able to make the best decisions of the three approaches. Based on these observations, we propose to use the full "cost-based" approach

for simple queries that involve no more than four collections and heuristics for more complex queries. In the future, we plan to investigate the tradeoffs of optimization time and application performance for some of the variants described in Section 3.4.

8 Acknowledgements

We thank Mary Tork Roth, Peter Schwarz and Bart Niswonger for their help with this work.

References

[ASU89] A. Aho, R. Sethi, and J. Ullman. *Compiler Construction*, volume II. Addison-Wesley, 1989.

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

[CDN+97] M. Carey, D. DeWitt, J. Naughton, M. Asgarian, J. Gehrke, and D. Shah. The bucky object-relational benchmark. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 135–146, Tucson, AZ, USA, May 1997.

[CK97] M. Carey and D. Kossmann. On saying "enough already!" in SQL. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 219–230, Tucson, AZ, USA, May 1997.

[CKV93] K. Curewitz, P. Krishnan, and J. 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.

[D+90] O. Deux et al. The story of O_2. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):91–108, March 1990.

[DFJ+96] S. Dar, M. Franklin, B. Jónsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 330–341, Bombay, India, September 1996.

[DHM+98] S. Deßloch, T. Härder, N. Mattos, B. Mitschang, and J. Thomas. KRISYS: Modeling concepts, implementation techniques, and client/server issues. *The VLDB Journal*, 7(2):79–95, April 1998.

[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.

[GK94] C. A. Gerlhof and A. Kemper. A multi-threaded architecture for prefetching in object bases. In *Proc. of the Intl. Conf. on Extending Database Technology (EDBT)*, volume 779 of *Lecture Notes in Computer Science (LNCS)*, pages 351–364, Cambridge, United Kingdom, March 1994. Springer-Verlag.

[HKU99] L. Haas, D. Kossmann, and I. Ursu. An investigation into loading a cache with query results. Technical report, IBM Almaden, San Jose, CA, March 1999.

[HKWY97] L. Haas, D. Kossmann, E. Wimmers, and J. Yang. Optimizing queries across diverse data sources. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 276–285, Athens, Greece, August 1997.

[HMNR95] T. Härder, B. Mitschang, U. Nink, and N. Ritter. Workstation/Server-Architekturen für datenbankbasierte Ingenieuranwendungen. *Informatik – Forschung und Entwicklung*, 10(2):55–72, May 1995.

[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.

[KJA93] A. Keller, R. Jensen, and S. Agrawal. Persistence software: Bridging object-oriented programming and relational databases. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 523–528, Washington, DC, USA, May 1993.

[KK95] A. Kemper and D. Kossmann. Adaptable pointer swizzling strategies in object bases: Design, realization, and quantitative analysis. *The VLDB Journal*, 4(3):519–566, August 1995.

[KKM98] A. Kemper, D. Kossmann, and F. Matthes. SAP R/3: a database application system. Tutorial handouts for the ACM SIGMOD Conference, Seattle, WA, USA, June 1998.

[Loh88] G. 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.

[Mos92] J. E. B. Moss. Working with persistent objects: To swizzle or not to swizzle. *IEEE Trans. Software Eng.*, 18(8):657–673, August 1992.

[PZ91] M. Palmer and S. 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.

[ROH98] M. Tork Roth, F. Ozcan, and L. Haas. Cost models DO matter: Providing cost information for diverse data sources in a federated system. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, Edinburgh, GB, September 1998.

[Rou91] N. Roussopoulos. The incremental access method of view cache: Concepts, algorithms, and cost analysis. *ACM Trans. on Database Systems*, 16(3):535–563, September 1991.

[RS97] M. Tork Roth and P. Schwarz. Don't scrap it, wrap it! A wrapper architecture for legacy data sources. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 266–275, Athens, Greece, August 1997.

[SAC+79] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access path selection in a relational database management system. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 23–34, Boston, USA, May 1979.