Query Execution Techniques for Caching Expensive Methods

Joseph M. Hellerstein* **Jeffrey F. Naughton**

University of Wisconsin, Department of Computer Sciences
1210 W. Dayton St., Madison, WI 53706
jmh@cs.berkeley.edu, naughton@cs.wisc.edu

Abstract. Object-Relational and Object-Oriented DBMSs allow users to invoke time-consuming ("expensive") methods in their queries. When queries containing these expensive methods are run on data with duplicate values, time is wasted redundantly computing methods on the same value. This problem has been studied in the context of programming languages, where "memoization" is the standard solution. In the database literature, sorting has been proposed to deal with this problem. We compare these approaches along with a third solution, a variant of unary hybrid hashing which we call *Hybrid Cache*. We demonstrate that Hybrid Cache always dominates memoization, and significantly outperforms sorting in many instances. This provides new insights into the tradeoff between hashing and sorting for unary operations. Additionally, our Hybrid Cache algorithm includes some new optimizations for unary hybrid hashing, which can be used for other applications such as grouping and duplicate elimination. We conclude with a discussion of techniques for caching multiple expensive methods in a single query, and raise some new optimization problems in choosing caching techniques.

1 Introduction

Object-Relational and Object-Oriented Database Management Systems allow users to register their own functions or "methods", and invoke these methods in declarative queries. User-defined methods can be arbitrarily time-consuming, so minimizing the amount of time spent in these methods is an important factor in improving the efficiency of query processing.

In previous work we have focused on query optimization techniques to appropriately place time-consuming ("expensive") predicate methods in a query plan [HS93, Hel94]. However, query optimization is only part of the problem; query execution techniques must also be enhanced to handle expensive methods, whether they appear in predicates (SQL's WHERE and HAVING clauses), or in other parts of a query (*e.g.* the SELECT, GROUP BY, and ORDER BY clauses.) In this paper we consider query execution techniques to minimize the impact of expensive methods wherever they appear in a query. In particular, we examine execution techniques that prevent a method from being computed more than once per query on the same input value.

As an example of the problem, consider the following SQL query, which displays a thumbnail sketch of photos used in advertisements:

```
SELECT thumbnail(product_image)
  FROM advertisements
 WHERE product_name = 'Brownie';
```

The `product_image` field is a reference to a multi-megabyte image object. The `thumbnail` method reads in this object, and produces a small version of the image. Reading and processing the object can be quite time-consuming, and hence `thumbnail` can be very expensive. Since a given product may have many different advertisement layouts with the same photo, `thumbnail` may need to be computed many times on the same image object. It would be wasteful to actually invoke `thumbnail` on each reference to an image. Instead, we would like to cache the result of `thumbnail` the first time it is called on a reference to an image. Then each subsequent time that `thumbnail` is called on a reference to the image, we could use the cached result instead of re-reading and re-processing the image.

The problem of redundant method invocation can even arise in queries where the method is invoked on a duplicate-free input, such as the key of a relation. This is because optimization techniques such as Predicate Migration [HS93] can postpone expensive predicate methods until after joins are performed. The join operation often produces duplicate copies of its input values; this is even possible if there are no duplicate values in either input. As a result, intelligent optimization of queries with expensive predicates further motivates intelligent execution techniques for such queries.

This paper explores query execution techniques to do method caching efficiently. We begin by considering algorithms to do caching for a single expensive method over relations of arbitrary size. We review two well-known caching algorithms — memoization and sorting — and present a third approach based on hybrid hashing, which we call Hybrid Cache. Hybrid Cache includes some modifications to traditional unary hybrid hashing, which can be applied to grouping and duplicate elimination as well as caching. We also discuss the interaction between these modifications and hybrid hash join techniques.

We implemented all three caching algorithms in the Illustra Object-Relational DBMS [Ill94], and we present a perfor-

*Current address: University of California, Berkeley, EECS Computer Science Division, 387 Soda Hall #1776, Berkeley CA 94720-1776. This work was funded by NSF grant IRI-9157357.

SIGMOD '96 6/96 Montreal, Canada
© 1996 ACM 0-89791-794-4/96/0006...$3.50

mance study of our implementation. The study demonstrates three main results:

1. The commonly proposed memoization technique is extremely inefficient for relations with many values, and its performance is dominated by Hybrid Cache.
2. Hybrid Cache is the most efficient technique for caching Boolean predicate methods.
3. There are tradeoffs between Hybrid Cache and sorting for non-predicate methods.

In our analysis of the tradeoffs between Hybrid Cache and sorting, we demonstrate that the cost of hashing is based on the number of *distinct values* in the input relation, while the cost of sorting is based on the number of *tuples* in the input relation. This makes hashing more attractive than sorting in many cases. However, we also find that the performance of hashing is dependent on the size of the method output, while the performance of sorting is not, which is why sorting can outperform hashing in some situations involving non-predicate methods. Although the similarities between sorting and hashing have been studied previously in some detail (particularly for joins), the results presented here highlight tradeoffs that had not been noted before.

We also present techniques to further accelerate queries with multiple expensive methods. This leads to a number of questions relating to the challenge of integrating method caching with query optimization. Although this paper focuses on query execution techniques only, we outline directions for further research in query optimization for caches.

1.1 Related Work

A increasingly large body of work has addressed the problem of query optimization in the face of expensive predicates ([CGK89], [YKY+91], [HS93], [Hel94], [KMPS94], etc.). These papers are orthogonal to this one in that they treat optimization issues, and assume that caching strategies are either efficient or non-existent.

Kemper, Moerkotte, Peithner and Steinbrunn have proposed "bypass" techniques for optimizing and executing disjunctive queries [KMPS94, SPMK95]. This work alleviates some of the problems of redundant computation in disjunctive queries, but does not present a solution to the problem of redundant method invocation in general — for example, it does not avoid duplicate method invocation for the simple `thumbnail` example above.

This paper focuses on techniques for caching and recovering method results "on the fly" while computing a query. A large body of orthogonal techniques provide *persistent caches*, which store method results in relations so that they can be reused across multiple queries over a period of time. Graefe provides an annotated bibliography of these ideas in Section 12.1 of his query processing survey [Gra93]. Persistent caches are akin to materialized views [GM95] or function indices [MS86, LS88] — from the point of view of a single query, they represent *precomputed* methods, rather than caches which are generated and used on the fly. Persistent caches are used in a way that is analogous to techniques for avoiding recomputation of common relational subexpressions [Sel88]. In order to generate a persistent cache, one may have to compute a method over an input relation with duplicate values; in this scenario, the techniques of this paper can speed the generation of the persistent cache. Conversely, if one already has a persistent cache available for a method, one may be able to circumvent the techniques presented in this paper. A mixture of techniques is possible as well: if one has an incomplete persistent cache, it can be used alongside the techniques of this paper. Our work is thus orthogonal to the work on persistent caching and common subexpressions; those approaches can coexist profitably with the techniques presented here.

Correlated SQL subqueries can be considered as a form of expensive method [HS93]. It has been demonstrated that the magic sets rewriting can be used to speed up such subqueries [MFPR90, SPL96], even in non-recursive SQL. This "magic decorrelation" avoids redundant computation by first computing the (duplicate-free) set of all input values to a correlated subquery, then feeding all the distinct input values into the subquery at once, and finally joining the result of all the subquery computations to the outer query block. This technique also allows the input to be handled in a set-oriented fashion, as opposed to the standard "tuple-at-a-time" invocation of a correlated subquery. Note however, that unlike subqueries, user-defined methods are necessarily invoked a tuple at a time. As a result, magic rewriting is not as effective for expensive methods as the techniques presented in this paper: the cost of forming the duplicate-free input set is equivalent to the cost of building a method cache, and on top of this cost magic rewriting requires an additional join and possibly also an additional materialization of the "supplementary" input [MFPR90]. For expensive subqueries, there are tradeoffs between magic and caching. Seshadri et al. propose new techniques for cost-based optimization of magic [SHP+96], which can be extended to the problem of choosing whether to use magic or caching for subqueries in a query plan. Discussion of using magic sets techniques for queries with multiple expensive functions appears in Section 6.3.

2 Background: To Cache or Not to Cache?

Methods with duplicate-free inputs derive no benefit from caching, and one should not pay for the overhead of caching in such scenarios. In some cases a query optimizer can ascertain that the input to a method in a plan is duplicate-free, and the optimizer can avoid caching, *e.g.*, when the input column(s) to a method form a primary key for the input relation. But there can also be scenarios that can go undetected by an optimizer, in which an input happens to have zero or very few duplicates, and the cost of caching outweighs its benefit.

We will see in this paper that an appropriately chosen caching technique always has a cost that is at most a fraction of an I/O per tuple. This is a small overhead to incur, especially as compared to the risk of recomputing an expensive method that may take the time of many I/Os. For this reason we do not consider cache replacement heuristics commonly used in operating systems and buffer managers (*e.g.*, LRU, clock, etc.) Instead, we always keep items in the cache long enough to ensure that we never recompute a previously cached value.

We will also see below that while an appropriately chosen caching technique is always relatively cheap, a poorly chosen caching technique can make caching extremely expensive.

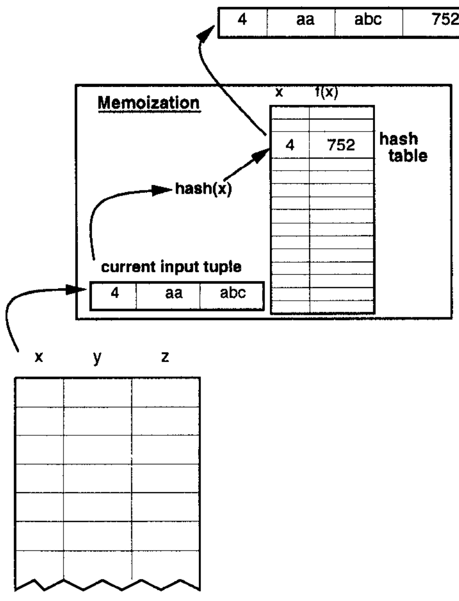


Figure 1: Memoization

So, to summarize, we recommend method caching in any situation where one cannot guarantee a duplicate-free input, but we also recommend that the technique used for caching be chosen carefully based on the analyses in this paper. We will return to these points in Section 7.

2.1 A Note on Method Semantics Over Time

One can cache functions indefinitely only if their results are independent of time. SQL3's `create function` command allows methods to be declared `variant` or `not variant`, with `variant` as the default [ISO94, Ill94]. A `variant` function is uncacheable, since it may return a different result on the same input at different times; a function that is `not variant` may be cached indefinitely. Typical time-independent data analysis and manipulation methods can be registered as `not variant` in an SQL3 system, and hence can be cached. One can refine these categories further, as was proposed for POSTGRES, by defining classes of methods whose results are cacheable over natural periods, such as the length of a transaction or a query[1] [Hel92].

2.2 Environment for Performance Study

Our performance study was run on a development version of Illustra, based on the publicly available version 2.4.1. Illustra was run with 256 buffers, and settings to produce traces of query plans and execution times. The machine used was a Sun Sparcstation 10/51 with 2 processors and 64 megabytes of RAM, running SunOS Release 4.1.3. One disk was used to hold the databases, another for Illustra's binaries, and a third was used for paging. Due to restrictions from Illustra, the performance times presented are relative rather than absolute: the graphs are scaled so that the lowest data point has the value 1.0. This scaling of results does not affect the conclusions of the experiments, which are based on relative performance. Graphs showing numbers of I/Os are not scaled.

[1]Note that SQL subqueries fall in this category: their results are cacheable for the length of a query and no longer, since subsequent operations in the same transaction may update the database, and thus affect the subquery's result.

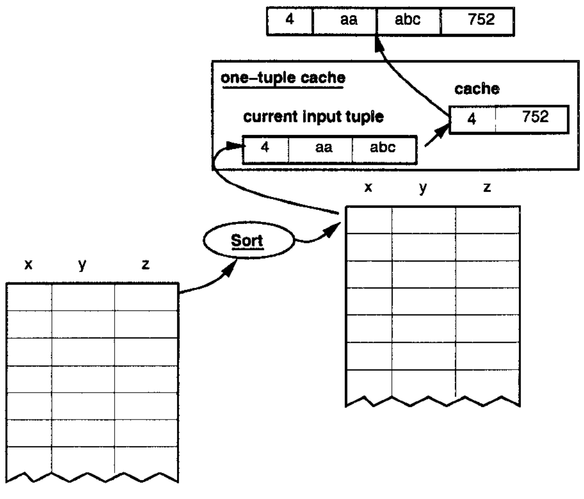


Figure 2: Sorting

3 Two Traditional Techniques

This section presents two previously proposed techniques for caching the results of expensive methods. Each of these algorithms (as well as the hybrid hashing algorithm below) can be used for expensive methods that appear anywhere in a query (`SELECT, WHERE, HAVING, GROUP BY, ORDER BY`, etc). The algorithms each take a relation as input and produce a relation as output, and therefore should be thought of as nodes in a plan tree, much like Join or Sort.

3.1 Main-Memory Hashtables: Memoization

A well-known technique for caching the results of computation is to build a hash table in memory. In the programming language and logic programming literature, this technique is often referred to as *memoization* [Mic68]. The algorithm is sketched in Figure 1. For our `advertisements` example, this hashtable would be keyed on a hash function of `product_image` values (which might be object identifiers or file names), and would store `(product_image,thumbnail)` pairs. During query processing, before computing `thumbnail` for a given `product_image`, the hashtable would be checked for the presence of that `product_image`. If a match were found, the resulting `thumbnail` would be taken from the hashtable. Otherwise, the `thumbnail` would be computed, and an entry would be made in the hashtable for the new `(product_image, thumbnail)` pair.

Memoization is easy to implement, and well-suited for workloads with small numbers of values. Note that it can work well even when many tuples are involved, since the processing per tuple is fairly simple. Unfortunately, the technique breaks down when faced with many values, since the main-memory hash table quickly becomes very large, and the operating system is forced to page it to disk. Since hash accesses by definition have low locality for distinct input values, the operating system paging schemes manage the memory very poorly. This is demonstrated dramatically in Section 5 below.

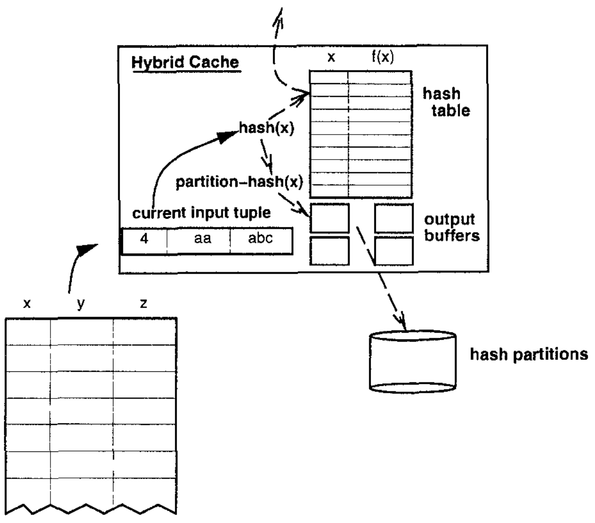


Figure 3: The staging phase of Hybrid Cache.

3.2 Sorting: The System R Approach

In [HS93] it was pointed out that SQL subqueries are a form of expensive method. The authors of the pioneering System R optimization paper [SAC+79] proposed a scheme to avoid redundant computation of correlated subqueries on identical inputs; their idea is directly applicable to expensive methods in general. They noted that

> if the referenced relation is ordered on the referenced column, the re-evaluation [of the subquery] can be made conditional, depending on a test of whether or not the current referenced value is the same as the one in the previous candidate tuple. If they are the same, the previous evaluation result can be used again. In some cases, it might even pay to sort the referenced relation on the referenced column in order to avoid re-evaluating subqueries unnecessarily [SAC+79].

In essence, the System R solution is to sort the input relation on the input columns (if it is not already so sorted), and then maintain a cache of *only the last input/output pair* for the method. This prevents redundant computation: after processing tuples with value x_0, if a new value x_1 appears there is no need to maintain the result of the method on x_0, since it is guaranteed that no more x_0s will appear. This is illustrated in Figure 2.

4 Hybrid Cache

The third technique we consider is unary hybrid hashing. Unary hybrid hashing has been used in the past to perform grouping for aggregation or duplicate elimination [Bra84], but to our knowledge this paper represents the first application of the technique to the problem of caching. Unary hybrid hashing is based on the hybrid hash join algorithm [DKO+84]. We introduce some minor modifications to unary hybrid hashing, and since we are applying it to the problem of caching we call our variant *Hybrid Cache*. We begin by outlining the basic Hybrid Cache algorithm; additional modifications to standard unary hybrid hashing are discussed further in Section 4.2.

The basic idea of Hybrid Cache is to do memoization, but to manage the input stream so that the main-memory hash table never exceeds a maximum size. This is accomplished by staging tuples with previously unseen input values to disk, and rescanning them later.

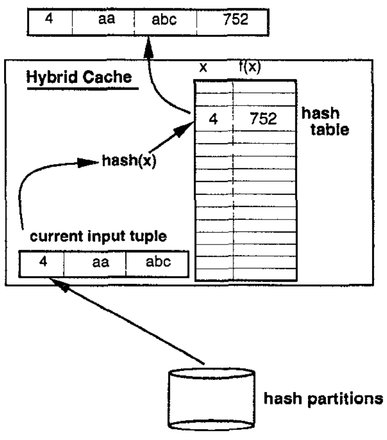


Figure 4: The rescanning stage of Hybrid Cache.

The Hybrid Cache algorithm has three phases. In the first or "growing" phase, tuples of the input relation are streamed in from some source (a base relation, a relational expression, etc.) As the tuples stream in, a main-memory hash table is developed and utilized exactly as in memoization. However, when the main memory hash table reaches a maximum size h (to be discussed in the next section), the algorithm enters its second phase, the "staging" phase. In this phase, each tuple of the input relation is considered as it arrives. If a match is found in the hash table, it is used just as in memoization. However, if no match is found, the input tuple is hashed again — via a different hash function than that used for the in-memory table — to one of a number of *partitions*, and placed in a buffer for that partition. The partition-hash function and number of partitions are chosen so that each partition has approximately the right number of *values* (not tuples!) to fill memory with a memoization hashtable (again, this derivation is given in the next section). When a partition's buffer fills, the buffer is written to the end of a contiguous area on disk allocated for that partition. The staging phase of Hybrid Cache is illustrated in Figure 3.

When the input relation is consumed in its entirety, the main-memory hash table is deallocated and the algorithm enters its third or "rescanning" phase. In this phase, the algorithm repeatedly chooses a new partition on disk, scans in the partition and handles the method invocations via naive memoization. When the partition is consumed, the hashtable is deallocated and a new partition is chosen. This process repeats until no partitions remain on disk. This stage of the algorithm is illustrated in Figure 4.

The Hybrid Cache algorithm is attractive for a number of reasons. First, it works at least as well as memoization for inputs of few values: all of the input can be handled in the growing phase, and the last two phases are not required. Second, like sorting it deals gracefully with inputs of many values, without requiring paging. Third, the amount of main memory needed for hashing is proportional to the number of *values* in the input, while the amount of memory needed for sorting is proportional to the number of *tuples* in the input. Thus hashing can often handle more input tuples in memory

variable	meaning
M	total pages of memory available
N	# of hashtable entries per page
h	# of pages to allocate to hashtable during growing/staging
b	# of pages to allocate to buffers during growing/staging
v	# of input values (estimated)

Table 1: Variables used in calculating memory allocation during the growing and staging phases of Hybrid Cache

than sorting can, which makes hashing faster than sorting for inputs with many duplicates. This is more apparent in Section 5, in which we present a performance study comparing the three techniques proposed so far.

Hybrid Cache provides one major advantage over traditional unary hybrid hash algorithms. In most hybrid hash algorithms, the first hashtable in memory is populated with all values which hash to partition 0. If there is skew in the parition-hash function, or if the number of values in the input is incorrectly estimated, partition 0 may be too large for memory. If this is the case, then the first hashtable will require paging. By contrast, in Hybrid Cache we grow the first hashtable so that it exactly fits in memory, while all hash partitions (including partition 0) are sent to disk. This modification guarantees that the first hashtable in memory will not grow too large. We will see in Section 5 that the cost of paging can be devastating; as a result, this minor modification to hybrid hashing can have major performance benefits. This technique can be directly applied to other applications of hybrid hashing as well, including grouping and duplicate elimination.

4.1 Memory During Growing and Staging

Two questions remain from the above description of Hybrid Cache:

1. How many partitions b should be allocated on disk?
2. How many pages of memory h can be used for the hashtable during the growing and staging phases?

The answer to the second question is actually a corollary to the answer to the first. During growing and staging, one page of memory must be allocated as a buffer for each partition; hence h is the memory left over after this allocation, *i.e.* if there are M total memory pages available, then $h = M - b$. This section presents derivations to choose b and h appropriately, so that they sum to M.

Given the (average) width of the input values and the output of the method, we can compute N, the number of hashtable entries that fit on a page. Ideally, each hash partition staged to disk should have as many values as make a hashtable that just fits in memory; *i.e.*, MN values. During the growing and staging phases, hN input values can be handled in the main-memory hashtable. Given v input values total, there should thus be

$$b = \frac{v - hN}{MN}$$

partitions on disk; each partition will hold MN of the $v - hN$ values that are not placed in the main-memory hash table during the growing phase. If we want to fill memory while scanning the input, we let $h + b = M$. Then by using some simple algebra, we can calculate h as a function of v:

$$h = M - b = M - \frac{v - hN}{MN} = \frac{M^2N - v}{N(M-1)}$$

To ensure that the buffers fit in memory, we require the constraint that $b \leq M$, which leads us via some additional algebra to the constraint that $v/N \leq M^2$. That is, the number of pages required to place all input values in a hashtable (with their outputs) must be less than the square of the size of memory. This is a well-known constraint on hashing; a similar constraint applies to sorting. These constraints can be overcome by recursive application of either partitioning (for hashing) or merging (for sorting) [DKO+84, Knu73]. Our implementation of Hybrid Cache in Illustra includes recursive partitioning to handle this situation.

The number of input values v can be estimated by using stored statistics [SAC+79] or via sampling [HOT88, HNSS95]. Unfortunately this estimation is subject to error, so we must consider how the algorithm will behave if estimates are imperfect. If v is estimated too high, h will be underestimated — *i.e.*, memory will be underutilized during the first two phases of Hybrid Cache by wasting memory on an excess of buffers. If v is estimated too low, the partitions on disk may contain too many values, and will require recursive partitioning on rescan. But note that if we have an upper bound on the number of input values (which is typical in most database statistics), v and the hash function can be chosen to ensure that repartitioning will never be required for relations within the memory constraints, since the number of values per partition can easily be capped. Thus the worst behavior of Hybrid Cache is that it may under-utilize memory during the first phase. Although this can cause unnecessary staging and rescanning of some tuples, it does not cause paging, nor does it require any tuple to be staged/rescanned more than once (as is required by recursive partitioning).

4.2 The Memoization Window: A Pathological Case

This section points out an additional optimization for unary hashing, which arose from our comparison of memoization and Hybrid Cache. Given the descriptions above of Hybrid Cache and memoization, there is an unusual scenario in which memoization is preferable to Hybrid Cache. We present a modification to Hybrid Cache which makes it match the behavior of memoization in this scenario.

Consider the situation in which the input relation has exactly enough values so that the memoization hashtable fills but does not overflow physical memory (*i.e.* $v = MN$). In this case, memoization performs no I/O, and is very efficient. Hybrid Cache is not as efficient in this scenario, since it must use b pages of memory for output buffers. As a result, Hybrid Cache cannot fit the entire set of input values into its hashtable during the growing phase, and some fraction of the values must be sent to partitions on disk. If there are many tuples with these values, Hybrid Cache can be noticeably slower than memoization. There is thus a small window where memoization is preferable to Hybrid Cache.

The Hybrid Cache algorithm can be modified to correctly handle this "memoization window". In the growing phase we allow Hybrid Cache to fill *all* of physical memory with the hashtable, rather than just filling h pages. When the hashtable is M pages big and a new input value is seen, we isolate b pages of the hashtable. For each input/output pair in this portion of the hashtable, we use the hash function to compute the partition on disk to which the input corresponds. Having computed the appropriate partitions for all such pairs, we sort the pairs by partition and then write the pairs to special *cache* regions of their associated partitions on disk. We then deallocate those b pages of the hashtable and begin the staging phase, using those pages for output buffers. Later, during the rescanning phase, we initialize the main-memory hashtable for each partition with the contents of its cache region before beginning to scan tuples from the partition.

In fact, this modification can provide performance benefits even for inputs that do not fall in the memoization window. Consider any input relation that requires staging to disk. During the growing phase, some input value i and its corresponding output is stored in one of the last b pages of the hashtable. Until the growing phase completes, any tuple with input value i can be passed to the output without staging. This is in contrast to the unmodified version of Hybrid Cache, which would have staged all tuples of input value i to disk. Note that this optimization comes at a low execution overhead, namely writing b pages of memory to disk, and later reading them back. It does requires a somewhat more complicated implementation, however, and as a result we did not implement it in Illustra.

4.3 Hybrid Cache and Hybrid Hash Join

Hybrid Cache treats its input in much the same way that hybrid hash join treats its "probing" relation. In fact, our implementation of Hybrid Cache in Illustra reused much of the existing hybrid hash join code for the probing relation. The main-memory hashtable of Hybrid Cache is analogous to the "building" relation in hybrid hash join, but note that Hybrid Cache does not suffer from the "well-known" problems of hybrid hash join:

1. In hybrid hash join, if some join value has duplicates in the building relation, then all tuples containing this value must be brought into main memory at once, possibly resulting in paging. In Hybrid Cache, even if some input value has duplicates it only requires one entry in the main-memory hashtable. The distinction is that hash join requires materializing *tuples* from the building relation in main memory, while Hybrid Cache materializes only *distinct values*.
2. Hybrid hash join is very sensitive to optimizer estimation errors. If it underestimates the number of values in the building relation, or chooses a poor hash function, the first partition of the building relation is likely to be too large and have to be paged, resulting in as much as a random I/O operation (seek and write) per tuple of the first partition of the probing relation [NKT88]. By contrast, Hybrid Cache never over-utilizes memory, since its first partition is dynamically grown to the appropriate size. Both algorithms can under-utilize memory for the first partition if estimates are incorrect, but this is less dangerous than over-utilizing memory, which results in paging.

Hybrid hash join can be modified to have the first partition of its building relation be grown dynamically, as we do for Hybrid Cache. This addresses the second problem above, but does not always solve the first problem, since duplicate tuples in the building relation need to be inserted into the hashtable even after memory has filled. Note however that duplicate-free inputs are common for primary-key joins and semi-joins, and hence the optimizations of Hybrid Cache may frequently be useful in hybrid hash join. More general (though somewhat complex) solutions to the problems of hybrid hash join have been proposed by Nakayama, *et al.* [NKT88].

4.4 Sort vs. Hash Revisited

In analyzing hashing and sorting, Graefe presents the interesting result that hash-based algorithms typically have "dual" sorting algorithms that perform comparably [Gra93]. However, we observe in this section that one of his dualities is based on an assumption that does not apply to the problem of caching. This highlights an advantage of using Hybrid Cache for caching, and also exposes an important distinction between hashing and sorting that was not noted previously.

Graefe distinguishes between hashing and sorting by observing that hashing utilizes an amount of main memory proportional to the number of tuples in the *output*, while sorting utilizes an amount proportional to the number of tuples in the *input*, which is likely to be larger. This is analogous to — but significantly different from — our earlier observation that hashing is *value-based*, while sorting is *tuple-based.* (Note that in the cases of grouping and duplicate elimination, these two analyses are identical, since the number of tuples in the output is equal to the number of values in the input). Graefe's analysis of the situation leads him to a modification of sorting via replacement selection [Knu73], which allows sort-based grouping and duplicate elimination to utilize an amount of memory closer to the number of tuples in its output rather than that in its input.

However, note that *the number of tuples in the input and output are identical for caching*: one output tuple is produced for every input tuple. Graefe attempts to make sorting competitive with hashing by getting the sort cost to be proportional to the number of tuples of the output. In the case of caching, this does not improve the performance of sort. Thus the operative distinction between sorting and hybrid hashing is not based on the number of input vs. output tuples; rather, the distinction is that hashing is value-based, while sorting is tuple-based. As a result, hashing is often preferable to sorting for the purpose of caching, as we proceed to demonstrate.

5 Performance Study

We implemented memoization and Hybrid Cache in our version of Illustra. Sorting code was already provided in the system to support sort-merge join and SQL's `ORDER BY` clause, so to use sorting for method caching we simply added code for a one-tuple cache over the sort node. We ran experiments using a synthetic table `T` of 2 million tuples, each having 14 integer-valued fields (56 bytes of user data per tuple).

5.1 Experiment 1: Effects of Duplicates

Our first set of experiments explore the behavior of memoization, sorting, and Hybrid Cache as the percentage of duplicate values in the input is increased. We consider queries of the form `SELECT * FROM T WHERE xfalse(T.c);` where `xfalse` is an expensive method that always returns FALSE. We use `xfalse` so that no tuples satisfy the selection

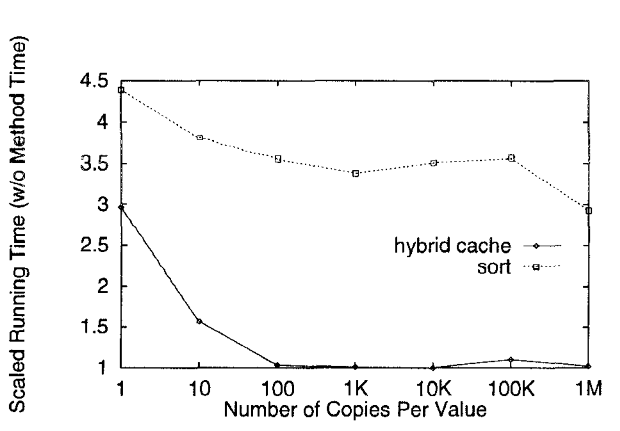


Figure 5: Running time of Hybrid Cache and sorting.

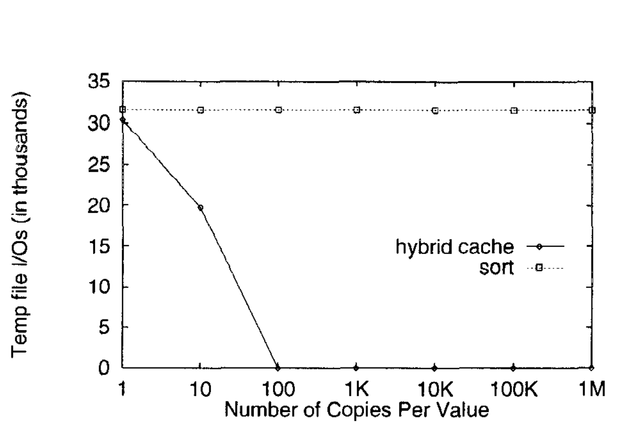


Figure 6: Temp-file I/O for Hybrid Cache and sorting.

predicate, and hence no time is spent propagating output. We ran this query 7 times per algorithm, and each time varied the column referenced by xfalse. The first column referenced had no duplicate values (*i.e.* 2 million distinct values), the second column had 10 copies of each value (*i.e.* 200,000 values), the third column had 100 copies per value (20,000 values), the fourth column had 1,000 copies per value (2,000 values), the fifth column had 10,000 copies per value (200 values), the sixth column 100,000 copies per value (20 values), and the final column had 1,000,000 copies per value (2 values). The table was sorted randomly so that duplicate copies were interspersed throughout each column. As in [Hel94], the "expensive" method we used did not actually do any time-consuming computation; however, its results were cached as if it were truly expensive. Each algorithm that we tested caused the method to be computed exactly once per value. Since the time spent in the method is the same across all algorithms, we do not include method computation time in the graphs below.

The results of the experiments are shown in Figure 5, which presents the running time of the query minus the time spent in xfalse. Figure 5 does not include a curve for memoization, since its performance was unacceptable when the hashtable did not fit in memory; operating system control over paging results in thrashing. In the case of 1 copy per value, the memoization technique ran for well over half a day, while the other techniques each took under a half an hour.[2] From 100 duplicates onward, memoization and Hybrid Cache behaved identically, since Hybrid Cache did not stage any tuples to disk, and memoization did not cause paging. In subsequent experiments we ignore memoization, since it is clearly not a general-purpose technique, and can never out-perform Hybrid Cache.

Figure 5 demonstrates that the performance of sorting is not competitive with that of Hybrid Cache. The main reason for this is illustrated in Figure 6, which shows the number of temporary-file I/Os that sorting and hashing require to stage and rescan tuples. This depicts what we expect from the preceding discussion: the I/O behavior of sorting is a function of the number of *tuples* in the input, while the I/O performance of Hybrid Cache — a hash-based algorithm — is a function of the number of *values* in the input. The running time of sorting does improve somewhat as the number of duplicates is increased, but this is due to speedups in the sorting of runs in memory. Even if the in-memory cost of sorting were always minimized (as it is for the 1M experiment), sorting would be about as efficient as the worst-case performance of Hybrid Cache (1 copy). As a result, even if sorting performed optimally in memory, Hybrid Cache would be the algorithm of choice for these experiments because of its I/O behavior. Like memoization, Hybrid Cache is optimized for inputs of few values, and like sorting it scales well to inputs with many values.

5.2 Effects of Output Size

The previous experiment shows Hybrid Cache out-performing sorting for an expensive predicate method. One important property of predicate methods is that their outputs are of Boolean type, and hence require only a few bits to represent. What if the output of a method is large? In the remaining experiments, we examine how this issue affects the performance of sorting and Hybrid Cache.

Expensive methods can appear outside of predicates. For example, they can be used in the SELECT, ORDER BY and GROUP BY lists of SQL queries. In such scenarios, the method outputs can be large. A typical example of such a method is decompress, which takes as input a reference to a large object, and produces as output an even larger object. Note that the output of decompress is a value, not a reference. As a result, the output is materialized in memory, and does not have an associated object or tuple identifier.

Our second set of experiments explores the performance of Hybrid Cache and sorting for methods producing outputs of various sizes. We consider queries of the form:

```
CREATE VIEW V AS
   SELECT xbig(T.c) FROM T;
SELECT xbig FROM V WHERE false(big);
```

where false is an inexpensive method that always returns FALSE, and xbig is an expensive method that returns an object that is 2 Kbytes big. The column T.c is varied among the differing numbers of copies, as in the previous set of experiments. We use false to prevent any tuples from needing to be passed to the output, and we disable Illustra's standard optimization of "flattening" the view into the outer query. As a result, xbig needs to be produced for every tuple of T.

Figure 7 presents the performance of sorting and Hybrid Cache for outputs of size 2 Kbytes. Unlike in our previous

[2]Actually, this performance result of memoization for the case of 1 copy per tuple is a lower bound on the running time, since the system eventually ran out of paging space and crashed. If our machine had been configured with more paging space, the query would have run for even longer before "completing".

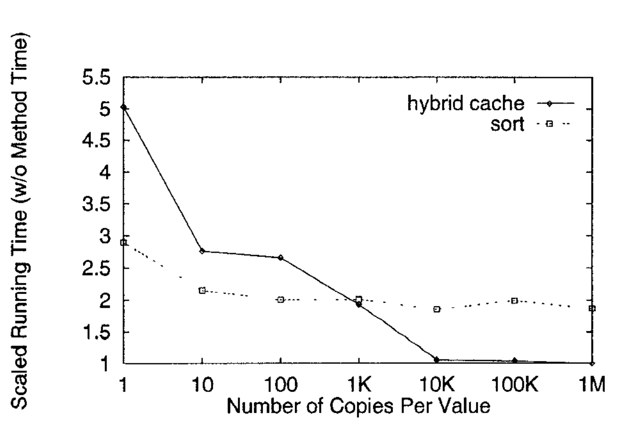


Figure 7: Running time of Hybrid Cache and sorting, 2K outputs.

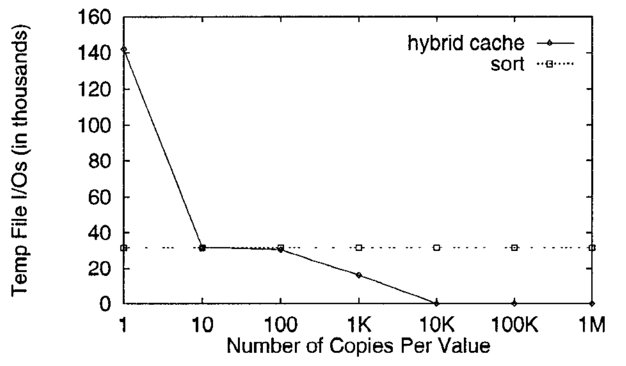


Figure 8: Temp-file I/O for Hybrid Cache and sorting, 2K outputs.

experiments, Hybrid Cache is no longer always superior to sorting; in fact, sorting is effective until the number of duplicates becomes quite high. Once again, the performance can largely be explained in terms of the I/O to temporary files, as shown in Figure 8. These graphs illustrate a second important distinction between sorting and Hybrid Cache: the I/O behavior of sorting is insensitive to the size of the output, but the I/O behavior of Hybrid Cache degrades when the output size is increased. This is because of the different ways that sorting and Hybrid Cache manage main memory. As illustrated in Figure 2, sorting with a one-tuple cache can utilize almost all of main memory for its input tuples, leaving just a single input/output value pair in memory. By contrast, Figures 3 and 4 illustrate that Hybrid Cache must share main memory among both inputs *and outputs* of the method. As output size is increased, Hybrid Cache can fit fewer and fewer entries into main memory. This means that more and more partitions are required to partition the input. One aspect of this problem is that very few values are placed in the hashtable during the growing phase, and as a result almost all tuples of the input must be staged to disk and subsequently rescanned.

A more significant aspect of this problem is that as output size is increased, the number of values per page (factor N of section 4.1) becomes smaller. As a result, the number of pages required for hashing more quickly approaches the limit of the square of the size of memory. When this limit is exceeded, recursive partitioning is required to avoid overflowing memory, and this requires extra I/O for recursively staging and rescanning the contents of partitions. This is illustrated in Figure 8, which shows the temporary-file I/Os required to stage and rescan tuples in both Hybrid Cache and sorting. In the case of one copy per value, Hybrid Cache is forced to do recursive partitioning, and must stage and restage tuples to disk multiple times. As the number of duplicates increases, Hybrid Cache no longer needs to do recursive partitioning, and can pass more and more tuples to the output during the growing phase, until at 10,000 copies per tuple every value fits in the hashtable during the growing phase (*i.e.*, Hybrid Cache behaves just like memoization).

The curve for Hybrid Cache in Figure 8 is curious, in that it levels off between 10 and 100 copies. This is explained by examining the number of tuples passed to the output during the growing phase of each query, as shown in Table 2. In the case of 10 copies, the number of tuples passed to the output (and hence never staged) represents just .55% of the input relation, and in the case of 100 copies, the number of tuples passed to the output represents only 5.45% of the input relation. In both cases, the fraction of the input that is not staged to the output is negligible, which explains why the two data points are approximately at the same height. However, since this percentage grows by a factor of 10 as we move right in Table 2, it quickly reaches 100% and prevents any input tuples from being staged.

Another curious point is that while Hybrid Cache and sorting do about the same number of temporary-file I/Os for 10 and 100 copies per value, sorting out-performs Hybrid Cache in terms of elapsed time for these inputs (Figure 7). The reason for this is that sorting requires fewer temporary files on disk then Hybrid Cache, since the Hybrid Cache partition size (MN) is small due to the large output size of the method (which causes N to be small). The time difference between Hybrid Cache and sorting is largely due to the cost incurred by Hybrid Cache in maintaining all its temporary files. Illustra allocates temporary files from the operating system, which places a limit on the number of open files allowed per process. As a result, when there are more temporary files than the operating system's open-file limit, Illustra must close and re-open temporary files to keep the number of open files below the limit. The difference in performance between Hybrid Cache and sorting at 10 and 100 copies is due to this overhead of closing and opening temporary files. Hybrid Cache could possibly be improved in this regard by having the DBMS manage its temporary files without using the file system.

5.3 Summary of Experimental Results

Our experiments demonstrate that there are two basic factors that affect Hybrid Cache and Sorting: the percentage of distinct input values, and the size of method outputs. Hybrid Cache handles duplicates well and large outputs poorly; sorting handles duplicates poorly and large outputs well.

This tradeoff is illustrated in Figure 9. An input relation is shown, along with a "virtual" column representing the result of the method computation. (For purposes of illustration, the input is depicted as if it were sorted on the inputs to the method.) The differently shaded areas represent the portions of the input materialized together in main memory by sorting and Hybrid Cache. Sorting materializes a large batch of input tuples and only one row of the virtual output col-

# of copies	1	10	100	1,000	10,000 and up
# of tuples	1,009	10,090	100,900	1,009,000	2,000,000
% of table	0.06%	0.55%	5.45%	54.5%	100%

Table 2: Number of tuples passed to the output during the growing stage of Hybrid Cache (2-Kbyte method outputs).

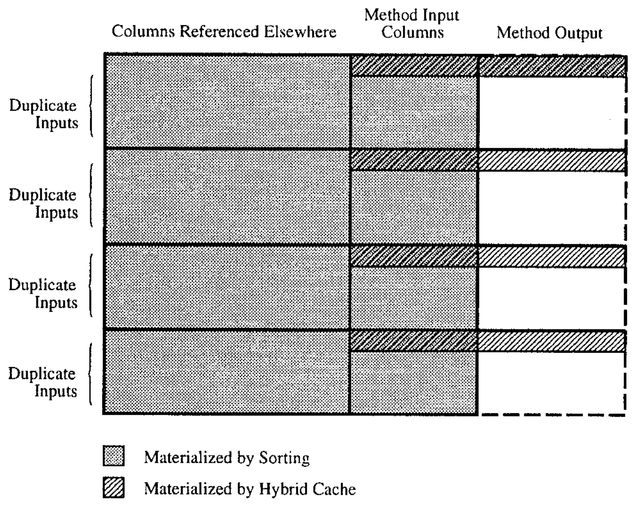


Figure 9: Portions of the input relation materialized together in memory.

umn. Hybrid Cache materializes only the distinct values of input columns that are referenced by the method, along with the corresponding output value for each row. The tradeoff is depicted spatially in Figure 9: the performance of the two algorithms varies with the size of the regions that they must materialize at once in memory. Whichever algorithm uses less memory at once can pass more tuples to the output without staging, and is less likely to require recursive I/O routines (partitioning or merging) to handle main-memory limitations.

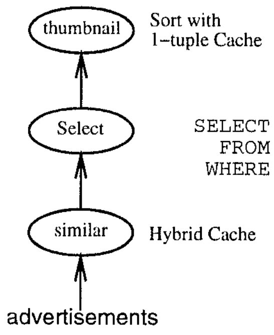
One more factor in performance is illustrated in Figure 9, but not in the experiments. Note that some columns of the input relation may not be required by the method; they may be needed for subsequent operators in a query plan. These columns must be materialized in memory for sorting, but not for hashing. This can be an additional advantage of hashing over sorting, resulting again from the fact that one sorts on tuples, but hashes only on input values.

As mentioned above, predicate methods have output values that are very small. As a result, Hybrid Cache is a better choice than sorting for expensive predicate methods. For other expensive methods, such as those in SELECT, ORDER BY, and GROUP BY lists, a choice between Hybrid Cache and sorting should be made based on the size of the distinct input values and method outputs, as compared to the expected memory needed for complete input tuples with duplicate input values. This decision can made fairly easily by using the usual cost formulas for non-recursive sorting and hashing; see, for instance, Graefe's query processing survey [Gra93].

6 Open Issues: Caching Multiple Methods

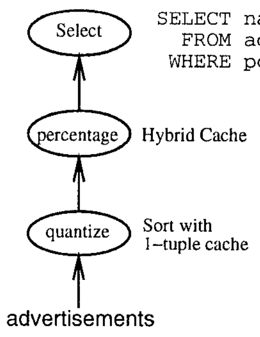
The previous sections present a detailed analysis and performance study of caching the results of a single method. A system that uses the appropriate choice of Hybrid Cache and sorting for each expensive method is already well-tuned for the sorts of queries possible in today's extensible systems. In this section we propose some additional techniques, which can be used in high-performance systems to further streamline the execution of some queries with multiple expensive methods. The techniques presented can all be beneficial, but the optimization issue of choosing between them remains an open problem beyond the scope of this paper. We present the techniques here, and defer discussion of the optimization problems to Section 7. We point out those techniques that can be achieved via query or method rewriting, so that systems without advanced optimizations can be coerced at user level into using these enhancements.

A query can contain multiple expensive methods, either in separate expressions (*e.g.* in two different select list expressions or predicates), or composed in a single expression. The caching techniques in the previous section can be naturally chained together to handle such scenarios. Methods in separate expressions may be handled naturally by a sequence of Hybrid Cache and sort nodes, as illustrated in Figure 10. A similar technique can be used for nested methods, as illustrated in Figure 11.



```
SELECT thumbnail(product_image)
  FROM advertisements
 WHERE ad_text similar '.*optic.*';
```

Figure 10: A query with 2 expensive methods.



```
SELECT name
  FROM advertisements
 WHERE pctage(quantize
              (product_image),'red') > 50
```

Figure 11: A query with nested expensive methods.

6.1 Sharing One Cache Among Multiple Methods

An important optimization to note is that *two methods with the same input variables can be computed by the same cache node.* Hybrid Cache and sorting both work by organizing their input tuples based on the input value to the method; methods with identical input values can thus be computed jointly. For example, the query of Figure 11 could be cached with a single Hybrid Cache node storing `(product_image, pctage(quantize(product_image), 'red'))` pairs.[3] This optimization is always beneficial for multiple methods with identical input variables.

Additional optimizations are possible when the input variables are similar but not identical, though some tradeoffs are involved which require optimization. Consider sorting first. Assume we have two methods f and g, where the set of input variables to f is I_f, and the set of input variables to g is I_g, and $I_f \subset I_g$. In this case, both methods may be cached by sorting the input tuples once on I_g, with the sort keys ordered so that elements of I_f precede those of $I_g - I_f$. For example, if the input to f was {`product_image`}, and the input to g was {`ad_text, product_image`}, the sorting should be done with on (`product_image, ad_text`) pairs. If I_f is not a proper subset of I_g but overlaps I_g significantly, we can still compute both with a single sort. We sort the input relation on $I_f \cup I_g$ with the sort keys ordered to have $I_f \cap I_g$ first. The in-memory cache for sorting will need to hold more than one input/output pair — in the worst case it will have to hold as many pairs as there are distinct values of the input for the same values of $I_f \cap I_g$.

Hybrid Cache can be modified to perform a similar optimization. Given two methods f and g with input variable sets I_f and I_g respectively such that I_f and I_g have a large intersection, the input can be hashed on $I_f \cap I_g$, but the inputs stored in the main-memory hashtables include all of the variables in $I_f \cup I_g$. This may result in hash collisions at the main-memory hashtable, which can degrade CPU performance if there are many distinct instantiations of I_g that have the same values for I_f. This technique can be used whether or not I_f is a subset of I_g.

Note that users can force two methods to be cached together by writing a new method which combines the two original methods into one. For nested methods (*e.g.*, $f(g(x))$, this simply entails writing a new method which is equivalent to the composition of the old methods (*e.g.*, $fg(x)$); for multiple methods that are not nested ($f(x), g(y)$), the new method must return a composite object containing the results of both of the methods. This observation can be handy for users of systems which do not implement cache-sharing.

6.2 Cache Ordering

When multiple expensive predicate methods appear in a query, the query optimizer orders them appropriately based on their relative costs and selectivities. However, when multiple expensive methods are nested in a single predicate, or when multiple expensive methods appear in a `SELECT` list, `ORDER BY` list or `GROUP BY` list, no selection takes place and even advanced optimization schemes will order the methods arbitrarily.

Yet there is often a choice of how to order cache nodes in a query plan. For example, methods from different `SELECT` list expressions can be ordered arbitrarily amongst themselves, as can methods that are at the same level of nesting in a nested expression (*e.g.*, g and h in the expression $f(g(x), h(y))$). A poor choice of ordering can be dangerous, even though no selection is involved, since cache nodes pass the outputs of their methods to the next operator in the pipeline. These outputs can be quite large, and the next cache node in the pipeline — whether it uses sorting or Hybrid Cache — may need to stage some of these large outputs to disk. It is therefore beneficial to order these cache nodes in increasing width of their output tuples. As an additional optimization, if the output of some cache node is very wide and multiple cache nodes follow it in the output, these outputs can be staged to disk once, referenced via pointers during the rest of the query plan, and rescanned only when they are next required.

6.3 Benefits of Magic for Cache Chains

A further, somewhat complex optimization is possible when a long chain of method caches appears in a query, and it is not advisable to merge the chain into fewer caches. Recall that the I/O cost of both sorting and Hybrid Cache results from staging tuples to disk and then rescanning them. To minimize this cost, we can first project the input to the caches so that only the columns referenced by the methods are passed into the caches; additionally, duplicates can be removed from this projection. The chain of caches can then be used to compute the methods for the distinct inputs, and the result of this chain can be rejoined with the original input relation to regenerate the projected columns and the correct number of duplicates. This is depicted in Figure 12 for the query of Figure 10. This technique is analogous to magic sets rewriting. The reason magic sets is beneficial in this scenario and not in the case of a single method (see Section 1.1) is that the cost of removing duplicates and joining can be amortized over the savings in staging and rescanning for multiple caches. A significant number of caches are required to recoup the cost of duplicate elimination and join, however. Choosing whether or not to perform this optimization is something that can be determined by a cost-based optimizer for magic [SHP+96].

7 Conclusions and Future Work

This paper studies the problem of avoiding redundant method invocation during query processing. Three algorithms are considered for caching method results: memoization, sorting, and a variant of unary hybrid hashing which we call Hybrid Cache. A performance study in Illustra demonstrates that memoization is ineffective in general, while Hybrid Cache dominates both of the other algorithms for expensive predicate methods. Non-predicate expensive methods can have arbitrary-sized outputs, and the study demonstrates that for methods with large outputs sorting is often preferable to Hybrid Cache. The variations we develop on hybrid hashing apply not only to caching, but also to other applications of unary hybrid hashing such as grouping for aggregation and duplicate elimination.

[3]Note that `'red'` in the above example is a constant, not a variable, and hence the input variables to the two methods are the same.

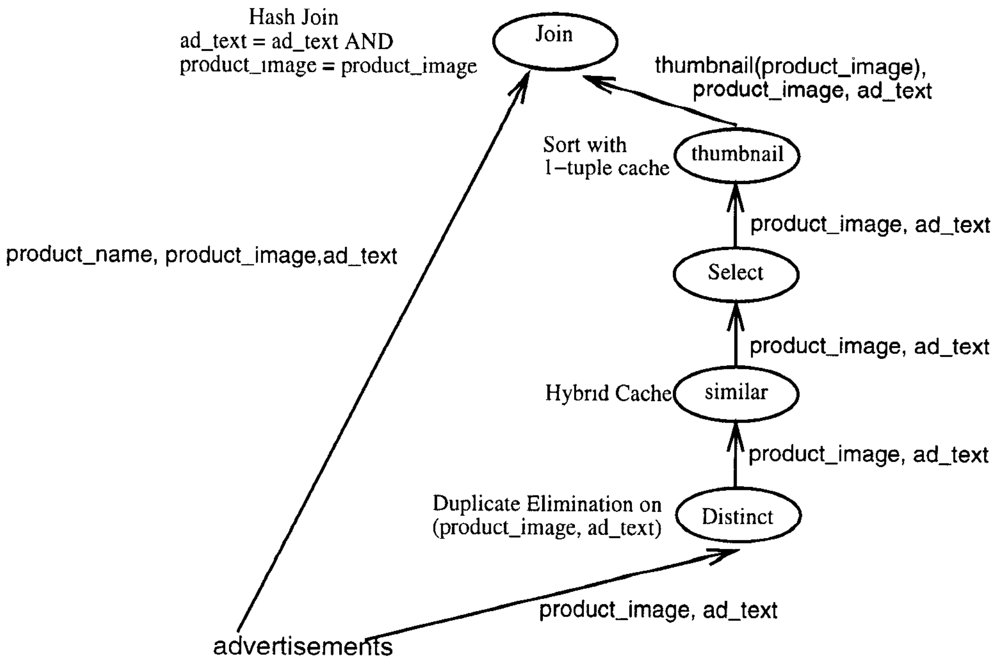


Figure 12: An alternative plan for the query of Figure 11.

We recommend that both sorting and Hybrid Cache be implemented in extensible Object-Relation and Object-Oriented database systems. It is attractive to simply use sorting for all expensive methods, since code for sorting is typically already present in most systems. We discourage this decision, however, since sorting is out-performed by Hybrid Cache for the common case of expensive predicates. If methods are always extremely expensive, then the difference between sorting and Hybrid Cache may often be overshadowed by method processing costs. But for methods which are only moderately expensive, appropriate use of Hybrid Cache will provide noticeable improvements in performance for inputs with many duplicates.

We also present a number of additional techniques for handling multiple methods in a query, which are designed to streamline the solution of chaining multiple Hybrid Cache and/or sort nodes together. Implementing these techniques is a second-order issue, since they are applicable only to complex queries with multiple expensive methods. For a high-performance extensible system, however, handling such queries as efficiently as possible may prove to be very important. As a first step, it is relatively simple to identify that multiple methods on the same input variables can be cached by a single operator. Further optimizations, especially those involving magic sets, require significantly more work and are much less generally applicable; as a result, they may only be appropriate for high performance systems. If necessary, database users can achieve these optimizations in more basic systems by rewriting their queries or methods.

Additional work is needed to clarify the relative importance of the techniques for multiple methods, and to capture the costs of caching appropriately in a query optimizer. A careful treatment of optimization and caching will require solutions to at least three new problems. First, the cost of caching needs to be captured in the optimizer and integrated with the predicate placement algorithm. The only previous optimization work which captured the presence of caching assumed that the cost of caching was negligible [Hel94]; this assumption may be acceptable in most cases, but this has yet to be determined. Second, the optimizer must be enhanced to choose which caching algorithm to use (if any) for different method inputs and outputs, taking into account issues of "interesting orders" [SAC+79]. Third, the optimizer must consider cache sharing, cache ordering, and magic sets techniques. As an additional challenge, it would be interesting to integrate the caching ideas in this paper with previous work on persistent caches, function indices, and common subexpression identification.

Acknowledgments

Many people provided comments and assistance with this work. Thanks to David DeWitt, Irwin Goldman, Goetz Graefe, Wei Hong, Yannis Ioannidis, Jussi Myllymaki, Raghu Ramakrishnan, Tom Reps and Praveen Seshadri. As usual, the staff of Illustra Information Technologies is to be thanked for its assistance, and for the opportunity to perform research on a commercial product.

References

[Bra84] Kjell Bratbergsengen. Hashing Methods and Relational Algebra Operations. In *Proc. 10th International Conference on Very Large Data Bases*, pages 323–333, Singapore, August 1984.

[CGK89] Danette Chimenti, Ruben Gamboa, and Ravi Krishnamurthy. Towards an Open Architecture for LDL. In *Proc. 15th International Conference on Very Large Data Bases*, pages 195–203, Amsterdam, August 1989.

[DKO$^+$84] David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael R. Stonebraker, and David Wood. Implementation Techniques for Main Memory Database Systems. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 1–8, Boston, June 1984.

[GM95] A. Gupta and I.S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Engineering Bulletin, Special Issue on Materialized Views and Data Warehousing*, 18(2):3–18, June 1995.

[Gra93] Goetz Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[Hel92] Joseph M. Hellerstein. Predicate Migration: Optimizing Queries With Expensive Predicates. Technical Report Sequoia 2000 92/13, University of California, Berkeley, December 1992.

[Hel94] Joseph M. Hellerstein. Practical Predicate Placement. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 325–335, Minneapolis, May 1994.

[HNSS95] Peter J. Haas, Jeffrey F. Naughton, S. Seshadri, and Lynne Stokes. Sampling-Based Estimation of the Number of Distinct Values of an Attribute. In *Proc. 21st International Conference on Very Large Data Bases*, Zurich, September 1995.

[HOT88] Wen-Chi Hou, Gultekin Ozsoyoglu, and Baldeao K. Taneja. Statistical Estimators for Relational Algebra Expressions. In *Proc. 7th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 276–287, Austin, March 1988.

[HS93] Joseph M. Hellerstein and Michael Stonebraker. Predicate Migration: Optimizing Queries With Expensive Predicates. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 267–276, Washington, D.C., May 1993.

[Ill94] Illustra Information Technologies, Inc. *Illustra User's Guide, Illustra Server Release 2.1*, June 1994.

[ISO94] ISO. ISO_ANSI Working Draft: Database Language SQL (SQL3) TC X3H2-94-331; ISO/IEC JTC1/SC21/WG3, 1994.

[KMPS94] Alfons Kemper, Guido Moerkotte, Klaus Peithner, and Michael Steinbrunn. Optimizing Disjunctive Queries with Expensive Predicates. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 336–347, Minneapolis, May 1994.

[Knu73] Donald Ervin Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley Publishing Co., 1973.

[LS88] C. Lynch and M. Stonebraker. Extended User-Defined Indexing with Application to Textual Databases. In *Proc. 14th International Conference on Very Large Data Bases*, pages 306–317, Los Angeles, August-September 1988.

[MFPR90] Inderpal Singh Mumick, Sheldon J. Finkelstein, Hamid Pirahesh, and Raghu Ramakrishnan. Magic is Relevant. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 247–258, Atlantic City, May 1990.

[Mic68] D. Michie. "Memo" Functions and Machine Learning. *Nature*, (218):19–22, April 1968.

[MS86] D. Maier and J. Stein. Indexing in an Object-Oriented DBMS. In Klaus R. Dittrich and Umeshwar Dayal, editors, *Proc. Workshop on Object-Oriented Database Systems*, pages 171–182, Asilomar, September 1986.

[NKT88] Masaya Nakayama, Masaru Kitsuregawa, and Mikio Takagi. Hash-Partitioned Join Method Using Dynamic Destaging Strategy. In *Proc. 14th International Conference on Very Large Data Bases*, pages 468–477, Los Angeles, August-September 1988.

[SAC$^+$79] Patricia G. Selinger, M. Astrahan, D. Chamberlin, Raymond Lorie, and T. Price. Access Path Selection in a Relational Database Management System. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 22–34, Boston, June 1979.

[Sel88] Timos Sellis. Multiple Query Optimization. *ACM Transactions on Database Systems*, 13(1):23–52, March 1988.

[SHP$^+$96] Praveen Seshadri, Joseph M. Hellerstein, Hamid Pirahesh, T.Y. Cliff Leung, Raghu Ramakrishnan, Divesh Srivastava, Peter J. Stuckey, and S. Sudarshan. Cost-Based Optimization for Magic: Algebra and Implementation. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 435–446, Montreal, June 1996.

[SPL96] Praveen Seshadri, Hamid Pirahesh, and T.Y. Cliff Leung. Complex Query Decorrelation. In *Proc. 12th IEEE International Conference on Data Engineering*, New Orleans, February 1996.

[SPMK95] Michael Steinbrunn, Klaus Peithner, Guido Moerkotte, and Alfons Kemper. Bypassing Joins in Disjunctive Queries. In *Proc. 21st International Conference on Very Large Data Bases*, Zurich, September 1995.

[YKY$^+$91] Kenichi Yajima, Hiroyuki Kitagawa, Kazunori Yamaguchi, Nobuo Ohbo, and Yuzura Fujiwara. Optimization of Queries Including ADT Functions. In *Proc. 2nd International Symposium on Database Systems for Advanced Applications*, pages 366–373, Tokyo, April 1991.